SUCESU

# INFORMÁTICA 82

XV CONGRESSO NACIONAL DE INFORMÁTICA
II FEIRA INTERNACIONAL DE INFORMÁTICA

# ANAIS

SUCESU

# INFORMÁTICA 82

XV CONGRESSO NACIONAL DE INFORMÁTICA
II FEIRA INTERNACIONAL DE INFORMÁTICA

# ANAIS

# PROJETO DE UM COMPILADOR PORTÁTIL PARA A LINGUAGEM "EDISON"

**Michael A. Stanton, Acir F. Martins,**
**Henrique M.G. de Aguiar, José Carlos P. das Neves**
Departamento de Informática, PUC/RJ
Rua Marquês de São Vicente, 225 – Rio de Janeiro, RJ

*Palavras-chave:* Compiladores, Edison, Microprocessadores, Portabilidade, Software básico.

O surgimento de compiladores de fabricação nacional e de microprocessadores criou uma demanda nacional para o desenvolvimento de software básico. Este trabalho descreve um projeto em execução na PUC/RJ que visa criar uma ferramenta para o desenvolvimento de software básico – a implementação portátil da linguagem Edison, recém-proposta por Brinch Hansen para microprocessadores. Depois de tecer considerações gerais sobre a implementação portátil de linguagens de programação são apresentadas as alternativas escolhidas neste projeto. O compilador adota uma estrutura multifásica, que permite concentrar (quase) toda a dependência da máquina objeto numa só fase. A implementação descrita se dirige aos microprocessadores 8086/8088 da Intel.

## 1. INTRODUÇÃO

Com a recente implantação de uma indústria nacional de fabricação de computadores e a disponibilidade crescente de microprocessadores de custo baixo, criou-se uma demanda para um tipo de software que não era desenvolvido no país – software básico, ou seja, sistemas operacionais, compiladores e utilitários em geral. Na falta de uma tradição de desenvolvimento deste tipo de software, a maior parte desta demanda tem sido satisfeita pela importação maciça de software básico. Esta situação continuará até que seja desenvolvida uma capacitação nacional nesta área. Esta capacitação só será alcançada com o uso de *ferramentas* adequadas para o desenvolvimento de software básico, e com a *formação* de profissionais adequadamente preparados a desenvolvê-lo.

O trabalho descrito aqui faz parte de um projeto de pesquisa em andamento na PUC/RJ que objetiva criar condições para o desenvolvimento de software básico para microprocessadores. A ênfase deste projeto é na criação de ferramentas, e, em particular, sistemas de desenvolvimento de software. Entendemos por sistema de desenvolvimento de software um ambiente acolhedor para a criação, compilação e execução de programas confiáveis de modo eficiente.

Estes requisitos de confiabilidade e eficiência (da produção) descartam de início o emprego de linguagem de montagem para desenvolver software básico. Sabe-se que o custo de uso desta linguagem é muito caro. Partimos portanto para o uso de uma linguagem de alto nível, e examinamos as características de diversas alternativas antes de selecionar uma delas para nossa linguagem de desenvolvimento.

Várias características foram levadas em conta na escolha da linguagem. Foram consideradas importantes (a) a adequação da linguagem aos microprocessadores que seriam usados, (b) o uso de conceitos modernos de abstração de dados, (c) a possibilidade de escrever programas concorrentes, tais como sistemas operacionais, (d) a possibilidade de compilação em separado, e (e) a simplicidade da linguagem.

As linguagens consideradas incluiam BCPL (RIC80), C (KER78), PASCAL Concorrente (BH77), Modula-2 (WIR80), Edison (BH81), Mesa (MIT79) e Ada (ADA81). BCPL e C foram rejeitadas por não satisfazerem os critérios (a) e (b). Mesa e Ada foram consideradas muito grandes. Não escolhemos PASCAL Concorrente pelo fato do seu autor ter proposto Edison como uma linguagem melhor. Preferimos portanto Edison e Modula-2. Cientes que já havia um projeto de implementar esta última linguagem na UNICAMP (KOW82), nossa escolha recaiu na linguagem Edison.

Outro aspecto importante do nosso projeto era o

ambiente de aplicação na PUC/RJ, onde pesquisas em hardware e software vêm sendo conduzidas no Laboratório de Engenharia de Sistemas de Computação (LESC), que junta os interesses mútuos dos Departamentos de Engenharia Elétrica e Informática. No LESC já havia experiência com microprocessadores de 8 bits tanto em hardware como em software. Já haviam iniciado os primeiros trabalhos em hardware com microprocessadores de 16 bits (Intel 8086/8088), e havia a necessidade de desenvolver software básico para estes equipamentos. Porém foi logo reconhecida a futilidade de dirigir nossos esforços somente para esta arquitetura, como muito bem poderíamos precisar de software básico para outras máquinas, ainda não conhecidas. Portanto, foi logo decidido que a implementação da linguagem Edison seria feita de tal forma que fosse facilmente dirigível para diversos microprocessadores. Isto é, a implementação seria portátil. Em segundo lugar, a primeira versão desta implementação seria dirigida para os microprocessadores 8086/8088 da Intel.

Este trabalho descreve mais detalhadamente o projeto da implementação portátil de Edison e sua versão para os 8086/8088 da Intel. Depois de apresentar umas considerações gerais sobre a implementação de uma linguagem e sua portabilidade (seção 2) são apresentadas as decisões estratégicas para nosso projeto (seção 3). A seção 4 apresenta a estrutura da parte analítica do compilador, que depende pouco da máquina alvo. Na seção 5 é apresentada a maneira de realizar a implementação para os microprocessadores 8086/8088 da Intel. Finalmente na seção 6 faremos alguns comentários sobre o projeto e sua execução.

## 2. PORTABILIDADE E A IMPLEMENTAÇÃO DE UMA LINGUAGEM

Nesta seção examinaremos o que necessita para implementarmos uma linguagem de programação.

Depois examinaremos as conseqüências para esta implementação de poder transportar facilmente para diversas máquinas alvo um mesmo programa feito na linguagem fonte.

### 2.1 Implementação de uma linguagem

Para podermos criar e executar software escrito numa linguagem de programação, são necessários vários componentes de software que se complementam. São estes o compilador, o suporte de execução e o ambiente de suporte de programação, que são descritos a seguir:

#### 2.1.1 O Compilador

A função básica do compilador é transformar o programa fonte, escrito na linguagem de programação em um programa objeto que possa ser executado pela máquina alvo da compilação (real ou virtual). Esta função pode ser decomposta em duas tarefas: análise e síntese (WAI 81).

*Análise:* Na tarefa de análise, o compilador faz o reconhecimento e a validação do programa fonte, indicando os erros nele encontrados, e transforma um programa válido, para uma representação interna equivalente ao original.

*Síntese:* Na tarefa de síntese, o compilador gera o código objeto, que é a representação do programa sendo compilado numa linguagem objeto que é reconhecida e executada pela máquina objeto, real ou virtual. Dizemos que a máquina objeto é real quando as instruções do código objeto são executadas diretamente pelo hardware (possivelmente microprogramado). A máquina virtual é simulada por software, isto é, o código objeto é interpretado por um programa que por sua vez executa numa máquina real.

#### 2.1.2 Suporte de execução

A maioria dos conceitos de uma linguagem de programação típica é mapeada pelo compilador de maneira natural em cima de facilidades presentes na linguagem objeto. Porém geralmente existem numa linguagem de programação certos conceitos fundamentais que não correspondem naturalmente a conceitos da linguagem objeto. Podemos citar como exemplos o endereçamento de variáveis dinâmicas, entrada/saída em termos de arquivos ou fluxos, programação concorrente e compilação em separado. No caso do endereçamento de variáveis dinâmicas, o compilador poderá impor a visão altamente estruturada da memória da linguagem de programação no espaço linear de endereços da maioria das máquinas objeto, através do uso de uma pilha de registros de ativação. Porém nos outros três casos citados, o compilador normalmente não tem o conhecimento necessário para traduzir totalmente o programa fonte. Nestes casos, ele gerará instruções para chamar rotinas para suprir estas deficiências. Estas rotinas estarão disponíveis em tempo de execução e constituem o que chamamos do *Suporte de execução* da linguagem fonte. Normalmente este suporte de execução é escrito numa linguagem diferente da linguagem compilada.

#### 2.1.3 Ambiente de suporte de programação

O desenvolvimento de software requer mais do que a existência de um compilador e suporte de execução. É necessário ter condições de criar programas fonte (através de um editor), guardar estes num sistema de arquivos, ter condições de depurar programas em teste, e gerenciar de maneira conveniente as diferentes versões dos diversos componentes que são combinados para compor uma aplicação. Na maioria dos casos, algumas ou todas estas facilidades são providas por um sistema operacional, não sendo necessariamente específicas a uma determinada linguagem de programação, mas servindo a várias. Podemos observar por outro lado que certos sistemas operacionais são muito especializados, oferecendo suporte somente para uma linguagem de programação.

Podemos citar como exemplo o sistema SOLO (BH 77), que provê um ambiente para o desenvolvimento de programas em PASCAL sequencial. Como parte do projeto ADA, foram recentemente especificadas as características que devem possuir um ambiente de suporte para programação em ADA (ADA 80). Com a presença deste ambiente, tornam-se quase invisíveis as características específicas da máquina objeto.

### 2.2 Portabilidade

A portabilidade de um programa se mede em termos do esforço necessário para transportá-lo para um ambiente diferente (computador, sistema operacional). Somente é prática se os programas forem escritos em uma linguagem abstrata, que esconde as diferenças entre computadores tanto quanto for possível. Programas

escritos numa mesma linguagem podem ser portáveis de várias maneiras: (i) – tendo diferentes compiladores para diferentes máquinas, o que somente é prático para linguagens largamente utilizadas;

(ii) – tendo um único compilador que pode ser modificado para gerar código para máquinas diferentes. Isto requer uma distinção dentro do compilador das partes de verificação, ou análise, do programa e da geração de código;

(iii) – tendo um "único computador" que pode ser simulado eficientemente em diferentes máquinas.

– O transporte do compilador pode ser feito de duas maneiras:

a) Modificando a geração de código para a outra máquina. Para isto devemos construir o compilador com duas partes bem distintas, a de análise e a de geração de código, de maneira que precisamos alterar somente a de geração. Com isto podemos mudar o compilador para uma máquina com arquitetura totalmente diferente.

b) Fazer o compilador gerar código para uma máquina virtual, que será interpretado na máquina real por um programa. Desta maneira precisamos apenas reescrever o interpretador para a outra máquina.

As duas maneiras acima descrevem, na realidade, como transportar os programas escritos na linguagem e não propriamente o compilador, pois o que temos é um compilador cruzado (cross-compiler) que compila programas em uma máquina e estes são executados em outra. Então para transportarmos o compilador em si, devemos reescrevê-lo na linguagem que ele compila. Assim ao compilarmos o próprio compilador obteremos um novo compilador que executa na máquina para o qual vamos transportá-lo.

### 2.3 A estrutura global de um compilador portátil

Existem 2 técnicas bastante difundidas na estruturação de programas. Uma delas se baseia na programação por refinamentos sucessivos e a outra se baseia na decomposição do programa em módulos. Vamos, neste item, mostrar as diferenças básicas destas duas técnicas, ressaltando suas vantagens e desvantagens quando o programa que se pretende desenvolver é um compilador que precisa ser compacto e portátil.

#### 2.3.1 Refinamento em passos sucessivos

Consiste na decomposição de tarefas em subtarefas e de dados em estruturas de dados, onde em cada passo do refinamento, uma ou várias instruções/especificações do programa são decompostas em instruções/especificações mais detalhadas. Essa decomposição termina quando todas elas são expressas em termos de uma linguagem de programação ou computador alvo. O grau de modularidade que se obtém com o uso desta técnica vai determinar a facilidade ou dificuldade com a qual um programa pode ser adaptado a mudanças (linguagem, computador) (WIR 71a).

No projeto de compiladores PASCAL, o uso desta técnica leva diretamente a um modelo descendente recursivo (WAI 81). Este modelo traz em si algumas desvantagens como a exigência da programação em linguagem recursiva, e de a geração de código ser espalhada através do compilador, sendo feita à medida que o programa vai sendo analisado, dificultando o transporte do compilador para outras máquinas. Outra desvantagem é o fato de que muitos procedimentos são ativados repetidamente por curto período de tempo o que impossibilita o uso de superposição de código sem sensível redução de desempenho.

A vantagem mais importante deste modelo, além de sua simplicidade, é o fato que com ele podemos manusear linguagens complexas sem custos catastróficos em tamanho ou velocidade (MCK 74). O compilador de PASCAL de WIRTH (WIR71b) utiliza esta estrutura.

#### 2.3.2 Decomposição Modular

Modularização é um mecanismo usado para promover a flexibilidade e compreensibilidade de um sistema e a diminuição do seu tempo de desenvolvimento (PAR 72). Na modularização as decisões de projeto devem ser feitas antes que o trabalho em módulos independentes tenha começado. Os benefícios esperados são:

– menor tempo de desenvolvimento em função de grupos separados poderem trabalhar paralelamente;

– possibilidade de fazer mudanças drásticas em um módulo sem a necessidade de mudar outros;

– possibilidade de estudar o sistema um módulo por vez, entendendo e projetando melhor o sistema.

A eficácia de uma modularização depende dos critérios usados na divisão do sistema em módulos, como:

a) – considerar cada passo maior do processamento como um módulo, baseado no fluxo de dados do sistema;

b) – esconder informações, de maneira que cada módulo seja caracterizado por seu conhecimento de uma decisão de projeto, que é escondida das outras, revelando somente sua interface. É proposto que se inicie com uma lista das decisões mais difíceis ou mais passíveis de mudanças.

## 3. UM COMPILADOR PORTÁTIL PARA EDISON

Nesta seção apresentamos as decisões estratégicas para nossa implementação de Edison. Os detalhes da implementação serão tratados nas seções 4 e 5.

### 3.1 Características da Linguagem (BH 81)

A linguagem Edison é derivada de PASCAL (JEN 75), com a qual compartilha muitas características, embora a sintaxe não seja sempre idêntica. As características semelhantes a PASCAL incluem:

- estrutura de blocos
- aninhamento de procedimentos e regras de escopo
- declaração de tipos estruturados
- comandos estruturados, com algumas generalizações
- recursão
- declarações deverão preceder o uso de identificadores.

As principais diferenças são as seguintes:

- uso de módulos (tipos abstratos)
- comandos concorrentes e de sincronização
- compilação em separado, e o uso de rotinas externas
- acesso a representação interna de dados
- ausência de rotinas de entrada/saída na definição da linguagem. (Uma implementação oferece facilidades primitivas de e/s através de rotinas particulares).

### 3.2 Estrutura geral do compilador

Para facilitar a portabilidade do compilador, adotamos a estrutura decorrente de decomposição modular da função de compilação em duas tarefas principais, análise e síntese (WAI 81). Assim uma alteração da máquina alvo afetaria quase somente a tarefa de síntese. A tarefa de análise foi ainda decomposta modularmente em quatro subtarefas mais simples de forma a permitir a fácil adoção de uma estrutura de múltiplos passos no caso de precisar adaptar o compilador para executar num microcomputador dispondo de pouca memória. Esta segunda decomposição segue aquela adotada por Hartmann (HAR 77), com ligeiras alterações. As subtarefas são: análise léxica, análise sintática, análise de nomes (escopos), e análise de declarações e de corpo. A estrutura detalhada da fase de análise será descrita na seção 4.

A tarefa de síntese consiste de uma única fase que tem por objetivo, gerar código, que, nesta primeira versão do compilador, usará linguagem de montagem para simplificar o compilador e facilitar sua verificação. Os detalhes da implementação que dependem da máquina objeto são discutidos na seção 5.

### 3.3 Interligação entre as fases do compilador

Cada fase do compilador recebe o programa em uma linguagem de saída. Com isto temos 6 linguagens envolvidas: a linguagem fonte (EDISON), as 4 linguagens intermediárias e a linguagem objeto da compilação.

Cada fase do compilador lê o programa na linguagem de entrada uma única vez, sequencialmente, gerando a saída também sequencialmente. Esta saída será a entrada da fase seguinte.

Toda informação que uma fase precisa está no programa de entrada, distribuída no código. Não se usam tabelas que sejam montadas em uma fase e consultadas em outra. Desta maneira as fases tem estruturas de dados internas independentes das estruturas internas das outras fases.

Como erros são detectados em várias fases de análise, suas indicações também são distribuídas no código. A primeira fase produz uma listagem do programa fonte e a última emite as mensagens de erros ocorridos em cada linha, independente da fase em que foram detectados.

A verificação sintática da linguagem fonte é feita uma única vez, na análise sintática. As fases posteriores recebem programas sintáticamente corretos, em forma de árvore linearizada, de maneira que a própria leitura equivale ao caminhamento na árvore, tendo apenas que identificar o "nó" sendo reconhecido.

### 3.4 A linguagem de implementação

Como foi discutida na seção 2.2 a melhor maneira de produzir um compilador portátil é escrevê-lo na linguagem que ele compila, no nosso caso, em Edison. Porém existe o tradicional problema inicial. Não tendo acesso a um compilador existente para Edison, não podemos usar esta linguagem imediatamente. A solução adotada foi do "half-bootstrap" (LEC 78), onde criaríamos primeiro um compilador de Edison escrito em PASCAL 8000 (uma implementação de PASCAL (JEN 75) para o IBM 370), que depois seria traduzido manualmente para Edison e compilado por si mesmo. Tanto a versão em PASCAL 8000, como a versão em Edison gerariam código para a máquina alvo (8086/8088 da Intel). Na figura 1 este processo é ilustrado usando o formalismo de Lecarme (LEC 78). Como já mencionado, o código objeto gerado usa a linguagem de montagem da máquina alvo, e precisa ser montado por software executado na máquina alvo.

## 4. A FASE DE ANÁLISE

### 4.1 Análise léxica

Esta primeira fase do compilador converte o texto fonte escrito em Edison para o primeiro código intermediário. Esta conversão consiste em ler o texto fonte caracter a caracter e transformá-lo em uma seqüência de inteiros representando identificadores, constantes e operadores. Cada categoria de símbolo começa com uma única classe de caracteres, desta forma o procedimento da análise léxica que classifica o caracter por grupo de símbolos será na realidade um comando CASE em que cada alternativa ativará o procedimento que tratará daquela categoria de símbolo em particular. Um identificador é colecionado caracter a caracter em uma cadeia, e é então aplicada uma função HASH para determinar um índice na tabela de símbolos. A seguir é feita uma pesquisa na tabela e é então verificado se este identificador já existe. Caso não exista, ele será inserido na tabela na posição determinada pela função HASH juntamente com um valor inteiro (nome) associado ao identificador no arquivo de saída. Os nomes padrões da linguagem são os primeiros a serem inseridos na tabela, no início da compilação. Todos os identificadores de mesmo nome terão o mesmo índice.
EXEMPLO:

- texto fonte

  B: = (A+C) * D;

- saída da análise léxica (seqüência de inteiros que representam)

  id(B) recebe abre id(A) mais id(C) fecha

  vezes id(D);

### 4.2 Análise sintática

A função desta fase do compilador é fazer a verificação das construções sintáticas da linguagem, convertendo a saída da análise léxica para uma notação pós-fixada. Nesta notação, a seqüência de operações fica bem determinada sendo assim eliminados operadores redundantes tais como parênteses em expressões e sinais de pontuação.

O analisador sintático utiliza uma gramática RRP LL(1) e será implementado baseado no uso de tabelas que representam os grafos sintáticos da linguagem. O modelo consiste basicamente do uso de uma pilha sintática (usada para armazenar partes reconhecidas de uma forma sentencial), uma pilha de estados do analisador e uma tabela de ações. No reconhecimento de um trecho de programa, o analisador verifica o símbolo na ENTRADA e o estado atual (no topo) da PILHA DE ESTADOS e consulta a TABELA DE AÇÕES. Nesta tabela estão indicados os procedimentos para cada estado/símbolo de entrada, os quais podem ser:

- o reconhecimento do símbolo de entrada com a

ação sintática de empilhamento e mudar o estado da pilha de estados.
- o reconhecimento do final de uma estrutura sintática, com ações de redução (desempilhamento) dos símbolos da estrutura já reconhecida, substituindo-os na pilha pelo não terminal que representa a produção.
- o reconhecimento de um erro sintático com a respectiva ação de recuperação, de maneira que o código intermediário de saída esteja sintaticamente correto.

Um campo da tabela de ações indica as ações semânticas a serem realizadas, que consistem da geração da estrutura correspondente na linguagem de saída (em um arquivo intermediário).

A escolha deste modelo foi fundamentada no fato do analisador se tornar mais compacto (utilizando estruturas de dados adequadas) e já termos disponível um gerador automático de tabelas (projeto NHÃONHÃO do programa de sistemas da COPPE/UFRJ). Este gerador fornece a tabela a partir da especificação da gramática da linguagem (TEL 81).

EXEMPLO:

- saída da análise léxica

  id(B) recebe abre id(A) mais id(C) fecha vezes id(D)

- saída da análise sintática

  nome(B) anome nome(A) Fnome valor nome(C)

  Fnome valor mais valor nome(D) Fnome vezes

  armazene

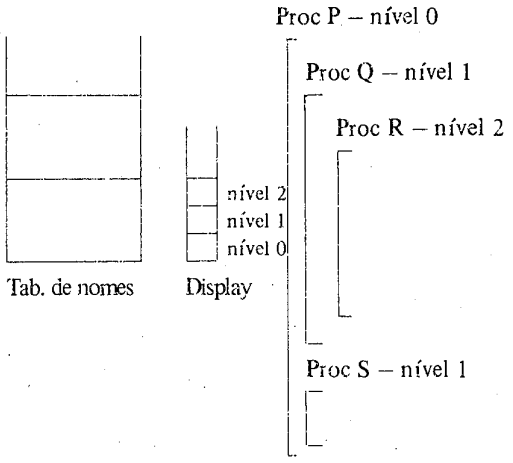
que equivale a BAC+D*:=

### 4.3 Análise de Nomes

A função da análise de nomes é associar índices distintos a identificadores iguais quando estes tiverem escopos diferentes (isto é, quando forem declarados em blocos diferentes), e assegurar as regras de escopo de EDISON. Exemplificando, em um texto fonte o analisador léxico converte todas as ocorrências do identificador "a" em id(n), onde "n" é um índice que corresponde ao identificador "a". É função da análise de nomes identificar as diferentes ocorrências de "a", convertendo-as em índices únicos. Desta forma as ocorrências de "a" declarado em uma procedure seriam convertidas para nome($n_1$) e de "a" declarado em outra procedure seriam convertidas, para nome($n_2$), onde $n_1$ e $n_2$ são índices distintos.

Esta conversão é feita através do uso de tabelas, que são construídas neste passo, à medida que nomes são introduzidos, e o uso do mecanismo de DISPLAY, que permite a identificação dos níveis de aninhamento de procedures e módulos.

EXEMPLO:




À medida que nomes (índices) vão aparecendo no texto, são pesquisados na tabela de nomes e inseridos, caso não encontrados. Cada vez que inicia um novo bloco, um novo nível de aninhamento é criado e um elemento do display aponta onde começa os nomes daquele nível, na tabela de nomes. Assim é possível fazer uma pesquisa no nível em que o nome é referenciado e em níveis que o englobam, assegurando as regras de escopo. Quando termina um bloco, a parte da tabela que corresponde ao seu nível é retirada, tornando a ser usado quando encontrar outro bloco de mesmo nível.

Quando um nome (índice) é inserido na tabela de nomes, a ele é associado um novo índice único, de maneira que um identificador de mesmo nome inserido em blocos diferentes passa a ter índices distintos. No código de saída são distribuídos os índices associados ao nome do identificador, e no caso de variáveis também o índice associado ao nome do tipo de variável. Uma referência a uma variável passa a ser da forma nome($n_1$,$n_2$), onde $n_1$ é o índice associado à variável e $n_2$ a seu tipo.

Para assegurar as regras de escopo de módulos, os identificadores exportados (que são declarados dentro do módulo, mas são visíveis no bloco, que o engloba diretamente) são considerados, nesta análise como se fossem declarados no bloco que engloba o módulo.

### 4.4 Análise de Declarações e de Corpo

A análise de declarações processa a parte de declarações do programa, analisando tipos e atribuindo endereços virtuais à variáveis, parâmetros e rotinas. Isto é feito construindo-se uma tabela de símbolos neste passo, de maneira que cada vez que um nome é declarado, uma entrada na tabela é criada. Outras referências ao nome são processadas através da tabela.

O processamento das declarações determina o endereço virtual dos nomes (variáveis, parâmetros e rotinas), isto é o nível de aninhamento e o deslocamento dentro deste nível. Através da análise dos tipos obtém-se o tamanho da variável para que possa ser gerado o deslocamento dentro do nível. Para isto, é necessário o conhecimento do tamanho dos tipos elementares na

máquina objeto, o que acarreta uma pequena dependência desta fase de análise à máquina. Esta dependência não compromete a portabilidade, podendo facilmente ser resolvida utilizando-se uma tabela externa ao compilador, que seria alterada ao transportar para outra máquina objeto.

Na análise de declarações módulos não tem nível de aninhamento próprio, sendo todas as suas variáveis alocadas no nível da procedure que a engloba, isto é, um módulo não tem um registro de ativação próprio, na pilha de execução.

Na análise do corpo dos procedimentos, é verificada a compatibilidade entre operandos e operadores, em função do tipo da variável. Operandos são compatíveis quando seus tipos têm o mesmo nome.

O código de saída deste passo é a interface entre a análise e síntese (geração de código). Ele apresenta uma estrutura que reflete a estrutura de blocos da linguagem, permitindo que possa ser construído um gerador de código para uma arquitetura ou outra, ou mesmo gerar código para um interpretador.

## 5. EDISON PARA OS 8086/8088 DA INTEL

Nesta seção discutiremos a parte da implementação que depende da máquina objeto. Esta é composta do gerador de código do compilador e o suporte de execução. A discussão destes será precedida por uma análise da máquina objeto e o modelo objeto que será adotado.

### 5.1 O hardware (INT 81)

Os microprocessadores 8086/8088 da Intel são de 16 bits, e podem endereçar até 1 Mb de memória. Em determinado momento o processador tem acesso a quatro "janelas" de 64 Kb endereçados relativos aos registradores de segmento (CS, DS, ES e SS). O programa sendo executado é endereçado relativo a CS, e a pilha de hardware relativo a SS. Dentro destas janelas, os modos de endereçamento de operandos incluem imediato, direto e indexado (simples e duplamente). Os modos indexados são de grande utilidade para a implementação de variáveis dinâmicas e arrays, numa linguagem como Edison.

Os tipos de dados de Edison correspondem naturalmente aos tipos nativos da máquina objeto. O tipo int corresponde a palavra de 16 bits para o qual tem operações aritméticas e de comparação. O tipo char corresponde ao byte de 8 bits que tem operações de comparação. Operações sobre o tipo set correspondem a operações lógicas e de comparação sobre bytes e palavras considerados como arrays de bits. As operações de atribuição e comparação dos tipos record, array e set correspondem a instruções que movem ou comparam cadeias de bytes ou palavras.

A máquina objeto está organizada em torno de um conjunto de 13 registradores de 16 bits, dos quais oito podem ser usados para operandos e resultados de operações aritméticas ou lógicas. Destes oito, quatro também servem como pares de registradores de 8 bits, para as operações sobre bytes, e quatro como registradores indexadores. Adicionalmente, vários destes registradores são usados implicitamente em certas instruções, como fonte ou destinação de um operando, para apontar fonte ou destinação de um operando, como ponteiro da pilha de hardware, ou como contador para uma iteração. Estas irregularidades nas propriedades dos registradores contribuem para dificultar a geração ótima de código para a máquina objeto.

### 5.2 O modelo objeto

O modelo usado para implementar Edison segue de perto aquele apresentado no capítulo 9 do livro de Barret (BAR 79), para o caso de Algol. As diferenças são devidas à ausência de arrays dinâmicos em Edison e a decisão de alocar na pilha o display. A memória é dividida em um segmento de dados (máximo tamanho de 64 Kb), e um segmento de código por procedimento compilado separadamente. Em Edison todas as variáveis são dinâmicas alocadas na pilha), e são naturalmente endereçadas relativas a SS. Para simplificar o código gerado, convencionamos que os outros registradores de segmento também apontariam o segmento da pilha (DS=ES=SS). Edison permite chamar procedimentos compilados separadamente, e também passar qualquer procedimento como parâmetro. Para evitar os problemas associados à relocação ficou resolvido que todo procedimento compilado em separado constituiria um segmento de endereços com origem zero, o que obriga uso da forma longa de chamadas de rotinas (inter segment call) para transferência de controle.

Uma conseqüência direta disto é a obrigação de usar a forma longa também para chamadas de rotinas no mesmo segmento, porque estas poderiam também ser chamadas de outro segmento, depois de serem passadas como parâmetros.

O suporte para endereçamento de variáveis supõe uma pilha de registros de ativação e um display de bases dos registros atualmente visíveis. Segue-se a prática usual de usar a pilha de hardware para os registros de ativação. Por falta de número adequado de registradores, somente os dois últimos níveis do display estão contidos em registradores (no caso BP e BX), sendo o resto do display alocado no registro de ativação. Considera-se que otimizando acesso ao nível local, e o próximo nível que o engloba, serve bem o uso característico de módulos.

O formato do registro de ativação é mostrado na figura 2.

Durante a execução de um comando concorrente, será necessário ter uma ramificação desta pilha de registros de ativação (a chamada pilha-cacto). Os ramos desta pilha serão alocados do espaço restante do segmento de dados, um para cada processo. Ao terminar cada processo, seu ramo será liberado.

### 5.3 Geração de código

Esta última fase do compilador recebe o programa analisado e convertido para uma representação equivalente em que toda a informação das declarações já foi distribuída no código.

Esta fase faz a transformação final para a linguagem de montagem da máquina objeto, que foi considerada a mais conveniente por enquanto, por facilitar a verificação do compilador.

Esta transformação envolve a interpretação do endereçamento de variáveis já gerado na fase anterior em termos do modelo apresentado na subseção 5.2, e a conversão da avaliação de expressões em notação pós-fixada para uma seqüência de operações envolvendo

o uso dos registradores do hardware. A otimização de alocação dos registradores apresenta a tarefa mais complexa devido a heterogeneidade destes registradores que já foi notada.

Finalmente, chamadas explícitas a rotinas padrão, tais como de entrada e saída, ou aquelas chamadas implicitamente para implementar os comandos concorrentes e de sincronização, serão convertidas ou chamadas indiretas ao pacote de suporte de execução, cujo endereço estará localizado em posição conhecida em tempo de execução.

### 5.4 Suporte de execução

O pacote de suporte de execução de Edison consiste de rotinas necessárias para a implementação de Edison, mas que não podem ser escritas nesta linguagem. Estas incluem a inicialização do ambiente de execução, a carga de programa, e a implementação de multiprogramação num único processador. Nesta implementação, estas rotinas serão escritas em linguagem de montagem, e na primeira implementação usarão as facilidades do sistema de arquivos de CP/M 86 (CPM81), particularmente para implementar a carga e ligação de procedimentos compilados em separado. Uma das rotinas do pacote será responsável para inicialização da execução de um programa em Edison, o que inclui colocar o endereço de entrada do pacote no segmento de dados do programa numa posição conhecida.

## 6. CONCLUSÕES

O projeto aqui apresentado foi iniciado em fevereiro de 1982, e esperamos ter a primeira versão do compilador funcionando no segundo semestre. Como a linguagem Edison serve para escrever sistemas operacionais, uma das primeiras aplicações será a criação de um ambiente de suporte de programação em Edison escrito na própria linguagem. Outra linha de ação já prevista é a aplicação na área de controle de processos, onde o suporte de execução será o mínimo necessário para a aplicação. Além dos autores deste relatório, dois outros alunos de mestrado estão trabalhando em aspectos destes problemas.

Agradecemos a colaboração de outros grupos de pesquisadores, e especialmente do Eng. Janvrot do CENPES/Petrobrás, e do Prof. Estevam de Simone da COPPE/UFRJ, que gentilmente permitiu-nos uso do seu gerador de tabelas de análise sintática. Agradecemos também o apoio financeiro da FINEP e do CNPq.

## BIBLIOGRAFIA

(ADA80) *Requirements for ADA Programming Support Environment (Stoneman)*, U.S. Department of Defense, 1980.

(ADA81) *ADA Reference Manual*, Springer Verlag, New York, 1981.

(BAR79) Barrett, W.A., e Couch, J.D. *Compiler Construction: Theory and Practice*, Science Research Associates, 1979.

(BH77) Brinch Hansen, P., *The Architecture of Concurrent Programs*, Prentice Hall, Englewood Cliffs, N.J., 1977.

(BH81) Brinch Hansen, P., "Edison – a multiprocessor language", *Software Practice & Experience*, 11(4), 325-361, 1981.

(CPM81) – *"CP/M-86 System Guide"*, Digital Research, Pacific Grove, CA. 1981.

(HAR77) – Hartmann, A.C., *A Concurrent Pascal compiler for minicomputers*, Springer Verlag, New York, 1977.

(INT81) – *iAPX 86,88 User's Manual*, Intel Corp., Santa Clara, CA., 1981.

(JEN75) – Jensen, K., e Wirth, N., *Pascal User Manual and Report*, Spring Verlag, New York, 1975.

(KER78) – Kernighan, B.W., e Titchie, D.M., *The C Programming Language*, Prentice Hall, Englewood Cliffs, N.J., 1978.

(KOW82) – Kowaltowski, T., e Pedro Jr., J., "Uma implementação da linguagem Modula-2 para o processador Intel 8086/8088", *1º Simp. Des. Software Básico p/Micros*, PUC/RJ, 1982.

(LEC78) – Lecarme, O. e Peyrolle-Thomas, M.-C., "Self-compiling compilers: an appraisal of their implementation and portability", *Software - Practice & Experience*, 8(2), 149-170, 1978.

(MCK74) – McKeeman, W.M., "Compiler Construction", em Goos, G. e Hartmanis, J., *Compiler Construction: an Advanced Course*, pp. 1-36, Springer Verlag, New York, 1974.

(MIT79) – Mitchell, J.G., Maybury, W., e Sweet, R., *"Mesa Language Manual, version 5.0"*, Xerox Corp., Palo Alto, CA. 1979.

(PAR72) – Parnas, D.L., "On the criteria to be used in decomposing systems into modules", *Comm. A.C.M., 15*, 1053-1058, 1972.

(RIC80) – Richards, M., e Whitby-Strevens, C., *"BCPL - the language and its compiler"*, Cambridge University Press, 1980.

(TEL81) – Teles, A.A., e de Simone, E.G., "Gerador de analisadores sintáticos RRP LL(1)", *Anais do VIII SEMISH*, 387-398, Florianópolis, S.C., 1981.

(WAI81) – Waite, W.M., e Carter, L.R., "An Analysis/synthesis interface for Pascal Compiler", *Software-Practice & Experience, 11*, 769-787, 1981.

(WIR71a) – With, N., "Program development by stepwise refinement", *Comm. A. C.M., 14*, 221-227, 1971.

(WIR71b) – Wirth, N., "The design of a Pascal Compiler", *Software-Practice & Experience*, 1, 309-333, 1971.

(WIR80) – Wirth, N., *"Modulat-2"*, Institut für Informatik, ETH Zurique, 1980.

Fig. 1 – "Half-bootstrap" do compilador Edison para o 8086/8088 da Intel. Notação de Lecarme (LEC78).

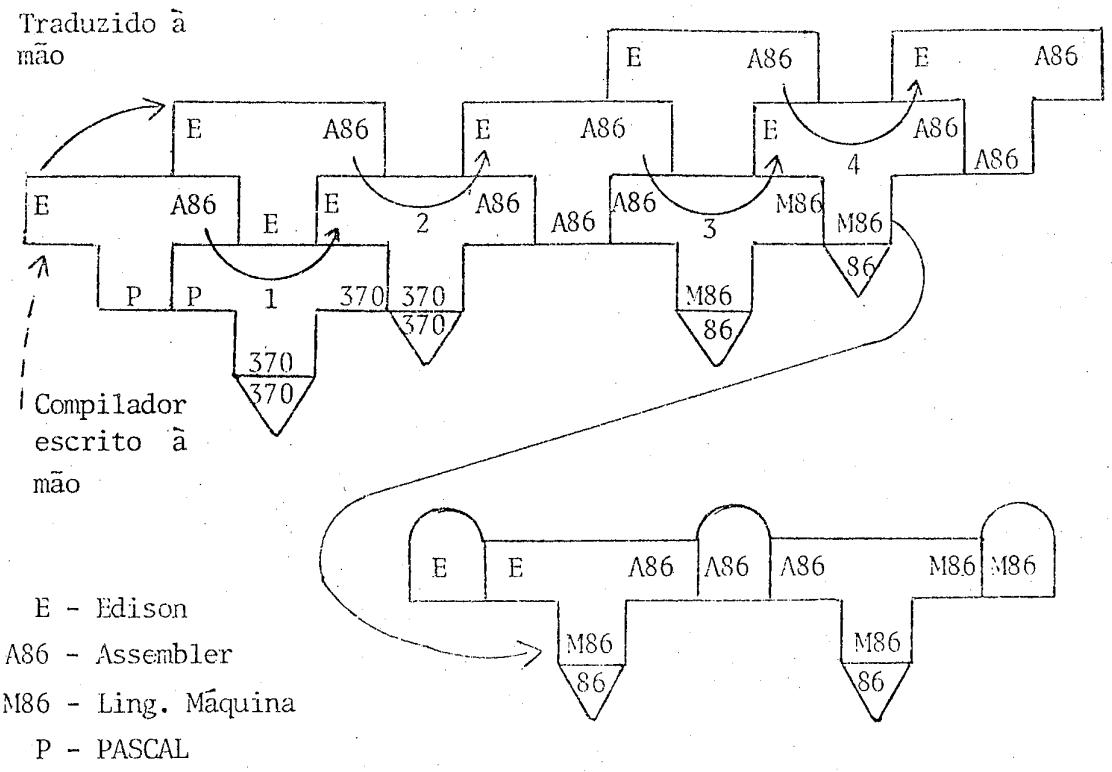



Fig. 2