Projeto de Compiladores (etapa 7):
`Pico`

Nicolas Maillard Claudio Schepke Stéfano Mór

Este documento detalha a especificação da sétima etapa do pequeno compilador `Pico`.

Chapter 1

Organização do projeto

A primeira etapa consiste na programação de duas estruturas de dados básicas. As etapas 2, 3 e 4 consistem na implementação dos analisadores lexicais e sintáticos. As etapas 5, 6 e 7 tratarão da fase de geração de código.

1. Estruturas de dados básicas (pilha e tabela de Hash). 1 ponto.
2. Analisador lexical (com auxílio do flex). 1 ponto.
3. Analisador sintático (LL(1). 1 ponto.
4. Análise sintática LR (Yacc). 1,5 ponto.
5. Análise semântica: declaração de variáveis, expressões. 1,5 ponto.
6. Análise semântica: controle de fluxo. 2 pontos.
7. Geração de código Assembly. 2 pontos.

O trabalho de programação da etapa 7 deve ser entregue até o dia 01 de Dezembro, 23h00 (segunda-feira). Será apresentado ao tutor no encontro do dia 02 de Dezembro, 10h30 ou no dia 04 de Dezembro. Ele receberá uma nota valendo 2 pontos do total da nota do projeto. A nota final do projeto valerá 50% da nota final da disciplina.

A programação deve ser feita no Linux, em C ANSI. Serõ usados, a partir da segunda etapa, as ferramentas Lex (flex) e Yacc (Bison). O último capítulo deste documento dá as primeiras instruções para usar o comando Make, no Linux, para compilar um programa. Também explica como se usa o Doxygen para documentar código.

É absolutamente fundamental que a especificação seja respeitada totalmente: os nomes dos procedimentos devem ser iguais ao especificado; os parâmetros devem ser conformes ao que se espera; etc. Os códigos devem ser comentados, de acordo com o padrão Doxygen. Encontra-se, nos programas entregues junto a essa especificação, exemplos de uso de Doxygen.

O projeto todo deve ser desenvolvido por grupos de 3 alunos. Faz parte do trabalho se distribuir a carga de trabalho de forma a fornecer uma implementação de boa qualidade: por exemplo, pode se "programar a quatro mãos" (dois alunos programam juntos), com o terceiro colega se encarregando de testes e da documentação. Pode-se também optar por uma distribuição dos procedimentos a ser programados. Aconselha-se que programação e teste não sejam feitos pela mesma pessoa. É interditado que um aluno nunca programe nada. É totalmente possível que os integrantes de um grupo tenham notas diferenciadas em função de seu envolvimento no trabalho.

Chapter 2

Especificação da Etapa 1

O código a ser implementado se encontra especificado em `Pico/src/Etapa1`. Pede-se implementar uma estrutura de pilha (ver o arquivo `stack.h`) e uma estrutura de tabela de símbolos (ver `symbol_table.h`). O diretório `doc/html/` contém essas especificações, legíveis on-line.

A pilha deverá suportar operações de `push` e de `pop` de elementos genéricos `void *`, ou seja referências sobre qualquer tipo se encontrarão na pilha. Outras operações clássicas são exigidas.

Uma tabela de símbolos é uma estrutura de dados interna a um compilador, que irá ser usada intensivamente nas outras etapas. Por ora, basta saber que se implementa através de uma tabela de Hash. Uma entrada na tabela de símbolos será caracterizada por vários campos (cujo detalhe está fora do escopo desta primeira etapa, ver o tipo abstrato `entry_t` em `symbol_table.h`), incluindo um campo "nome" de tipo `char*` (ou string, em C). A partir deste nome, deve-se calcular um hash para associá-lo a um número inteiro, que irá servir para acessar um vetor. Se houver colisões na função de Hash, cuidados devem ser tomados para desempatar. As funções exigidas são de inserção de uma entrada, de consulta, além de funcionalidades de verificação e de impressão do conteúdo de uma tabela de símbolos.

Salienta-se que se espera implementações *corretas*: sem perda de memória (lembrar que C não usa coletor de lixo: caso se use um `malloc`, deve-se usar um `free` em algum outro lugar), sem acesso fora da área de memória alocada, sem efeito colateral não controlado, etc...

A seguir, detalha-se o perfil de cada um dos procedimentos exigidos.

2.1 Referência do Arquivo stack.h

Funções

- void init_stack (stack *s)

  Inicializar a pilha. Uso tipico: init_stack(&minha_pilha);.

- void free_stack (stack *s)

  Liberar a memoria usada pela pilha. Uso tipico: free_stack(&minha_pilha);.

- int empty (stack s)

  Testar se a pilha esta vazia.

- int push (stack *s, void *item)

  *Empilhar um elemento na pilha. (O tipo do elemento é void *.).*

- void * pop (stack *s)

  Desempilhar o elemento no topo da pilha.

- void * top (stack s)

  Consultar o elemento no topo da pilha.

Variáveis

- typedef stack

  Aqui, voce deve completar a parte entre o 'typedef' e o 'stack' para inserir sua implementacao da estrutura de dados abstrata de pilha.

2.1.1 Descrição Detalhada

Versão:

1.1

2.1.2 Funções

2.1.2.1 int empty (stack *s*)

Testar se a pilha esta vazia.

Testa se a pilha esta vazia.

Parâmetros:

s uma pilha

Retorna:

- 0 se a pilha nao esta vazia,
- um valor diferente de zero se a pilha esta vazia.

2.1.2.2 void free_stack (stack * *s*)

Liberar a memoria usada pela pilha. Uso tipico: free_stack(&minha_pilha);.

'free_stack' eh o destrutor da estrutura de dados de pilha. Deve liberar qualquer espaco na memoria que tenha sido alocado para a implementacao interna da pilha passada em argumento. Um acesso a uma pilha, depois da chamada a 'free_stack' levara a um erro na execucao.

Parâmetros:

s um ponteiro sobre uma pilha (stack*).

Retorna:

nada (void).

2.1.2.3 void init_stack (stack ∗ *s*)

Inicializar a pilha. Uso tipico: init_stack(&minha_pilha);.

Inicializa a pilha, allocando qualquer espaço na memoria que seja necessario. Nao se deve efetuar nenhuma hipotese restritiva quanto ao numero total de entradas que podera conter a pilha num dado instante. 'init_-stack' devera ser chamado pelo usuario deste estrutura de dados, antes de poder usa-la. Qualquer referencia anterior que ele possa fazer a uma pilha tera comportamento nao-especificado.

Parâmetros:

s um ponteiro sobre uma pilha (stack∗).

Retorna:

nada (void).

2.1.2.4 void∗ pop (stack ∗ *s*)

Desempilhar o elemento no topo da pilha.

Desempilha o elemento no topo da pilha, e retorna-o. Remove este elemento da pilha.

Parâmetros:

s,um ponteiro sobre a pilha de onde se deve tirar um elemento.

Retorna:

o elemento que foi desempilhado, ou NULL se nao tinha como desempilhar alguma coisa.

2.1.2.5 int push (stack ∗ *s*, void ∗ *item*)

Empilhar um elemento na pilha. (O tipo do elemento é void ∗.).

Empilha um elemento na pilha.

Parâmetros:

s,uma referencia sobre a pilha onde se deve inserir o elemento.

item,uma referencia sobre o elemento a ser inserido.

Retorna:

0 se a insercao deu certo.

2.1.2.6 void∗ top (stack *s*)

Consultar o elemento no topo da pilha.

Retorna um ponteiro sobre o elemento no topo da pilha. Nao remove este elemento da pilha.

Parâmetros:

s,a pilha de que se deve consultar o topo.

Retorna:

o elemento, ou NULL se nao tinha nada no topo.

2.2 Referência do Arquivo symbol_table.h

Estruturas de Dados

- struct entry_t

Funções

- int init_table (symbol_t *table)
  Inicializar a tabela de Hash.
- void free_table (symbol_t *table)
  Destruir a tabela de Hash.
- entry_t * lookup (symbol_t table, char *name)
  Retornar um ponteiro sobre a entrada associada a 'name'.
- int insert (symbol_t *table, entry_t *entry)
  Inserir uma entrada em uma tabela.
- int print_table (symbol_t table)
  Imprimir o conteudo de uma tabela.
- int print_file_table (FILE *out, symbol_t table)
  Imprimir o conteudo de uma tabela em um arquivo.

Variáveis

- typedef symbol_t
  Encapsulacao de um tipo abstrato que se chamara 'symbol_t'.

2.2.1 Descrição Detalhada

Versão:

1.1

2.2.2 Funções

2.2.2.1 void free_table (symbol_t * *table*)

Destruir a tabela de Hash.

'free_table' eh o destrutor da estrutura de dados. Deve ser chamado pelo usuario no fim de seu uso de uma tabela de simbolos.

Parâmetros:

table,uma referencia sobre uma tabela de simbolos.

2.2.2.2 int init_table (symbol_t * *table*)

Inicializar a tabela de Hash.

Parâmetros:

table,uma referencia sobre uma tabela de simbolos.

Retorna:

o valor 0 se deu certo.

2.2.2.3 int insert (symbol_t * *table*, entry_t * *entry*)

Inserir uma entrada em uma tabela.

Parâmetros:

table,uma tabela de simbolos.
entry,uma entrada.

Retorna:

um numero negativo se nao se conseguiu efetuar a insercao, zero se deu certo.

2.2.2.4 entry_t* lookup (symbol_t *table*, char * *name*)

Retornar um ponteiro sobre a entrada associada a 'name'.

Essa funcao deve consultar a tabela de simbolos para verificar se se encontra nela uma entrada associada a um char* (string) fornecido em entrada. Para a implementacao, sera necessario usar uma funcao que mapeia um char* a um numero inteiro. Aconselha-se, por exemplo, consultar o livro do dragao (Aho/Sethi/Ulman), Fig. 7.35 e a funcao HPJW.

Parâmetros:

table,uma tabela de simbolos.
name,um char* (string).

Retorna:

um ponteiro sobre a entrada associada a 'name', ou NULL se 'name' nao se encontrou na tabela.

2.2.2.5 int print_file_table (FILE * *out*, symbol_t *table*)

Imprimir o conteudo de uma tabela em um arquivo.

A formatacao exata e deixada a carga do programador. Deve-se listar todas as entradas contidas na tabela atraves de seu nome (char*). Deve retornar o numero de entradas na tabela. A saida deve ser dirigida para um arquivo, cujo descritor e passado em parametro.

Parâmetros:

out,um descrito de arquivo (FILE*).
table,uma tabela de simbolos.

Retorna:

o numero de entradas na tabela.

2.2.2.6 int print_table (symbol_t *table*)

Imprimir o conteudo de uma tabela.

A formatacao exata e deixada a carga do programador. Deve-se listar todas as entradas contidas na tabela atraves de seu nome (char∗). Deve retornar o numero de entradas na tabela.

Parâmetros:

table,uma tabela de simbolos.

Retorna:

o numero de entradas na tabela.

2.2.3 Variáveis

2.2.3.1 typedef symbol_t

Encapsulacao de um tipo abstrato que se chamara 'symbol_t'.

Voce deve inserir, entre o 'typedef' e o 'symbol_t', a estrutura de dados abstrata que voce ira implementar.

Chapter 3

Especificação da Etapa 2

Essa segunda etapa do projeto consiste na implementação do analizador lexical (scanner), que deverá retornar os tokens reconhecidos num arquivo de entrada.

3.1 Explicações

A etapa consiste na definição, com o Flex, dos tokens que serão reconhecidos pelo compilador `Pico`. Cada token será implementado, em C, por um número constante inteiro através de diretivas `#define`. Por exemplo, `#define IDF 100` pode servir para definir o token `IDF`, que será encodificado no analizador lexical (e no futuro analizador sintático) através da constante 101.

A lista inteira dos tokens que deverão ser definidos, junto com a especificação (em português) dos lexemas que deverão ser reconhecidos como representando esses tokens, se encontra a seguir na próxima seção.

Para cada token, você deverá:

1. Defini-lo através de um `#define`. Essa lista de `define` deverá ser programada em um arquivo separado, limitado a este conteúdo, chamado de `tokens.h`. O seu analizador lexical deverá incluir `tokens.h` através de um:
   `#include ''tokens.h''`

   Os arquivos entregues em Pico/src/Etapa2 incluem um esqueleto para `tokens.h` e a inclusão do mesmo no arquivo de entrada usado com o Flex (chamado `scanner.l`).

2. Definir uma expressão regular, de acordo com a sintaxe do Flex, que especifique os lexemas representando esses tokens. Essas expressões regulares deverão se encontrar no arquivo `scanner.l`, usado como entrada do Flex. Para determinar essas expressões regulares, pode-se consultar o documento encontrado no Moodle, encontro 4, página 7. O arquivo `scanner.l` inicial, encontrado em Pico/src/Etapa2, contém um esqueleto "pronto a compilar".

3. Associar a essa expressão regular uma ação em C, que será acionada quando o analizador lexical irá reconhecer este token. A ação, em geral, será trivial e consistirá apenas no retorno (`return`) do token identificado. Eventualmente, será necessário executar uma ação mais complexa, de acordo com a especificação. Caberá, então, programar (em C) o código necessário ou usar estruturas de dados já prontas.

Exemplo: seja a especificação (irrealista) seguinte: o token `INT` deve ser retornado ao reconhecer os lexemas `integer` ou `int`. Deve-se programar, por exemplo:

```
#define INT 3333
%%
(integer|int)    { return( INT ); }
```

A linha do `#define` irá no arquivo `tokens.h`. A linha com a expressão regular, simples neste caso, e a ação em C (`return()`), deverá ser inserida no arquivo `scanner.l`.

Seu analizador lexical, chamado `pico`, deverá ler sua entrada a partir de um arquivo especificado em argumento na linha de comando. Ele deve usar uma tabela de símbolos para armazenar os identificadores (`IDF`) reconhecidos.

O arquivo `scanner.l` e o Makefile fornecidos em Pico/src/Etapa2/ são pontos de partida para escrever o analizador lexical e compilá-lo. Basicamente, basta complementar `scanner.l` e `tokens.h`. `make` ou `make pico` deve invocar o flex, gerar o arquivo C que implementa o analizador, e compilá-lo para obter o executável `pico`. Observa-se que o `scanner.l` já vem com um `main` pronto, o qual se encarrega de ler o arquivo em entrada e de chamar o scanner nele.

3.2 Especificação dos tokens

Os tokens são definidos em duas partes: os chamados "tokens simples" representam apenas um lexema (string) único, o que significa que a expressão regular os definindo é trivial (mas precisa ser implementada). Os outros tokens necessitam de expressões regulares menos imediatas, porém todas discutidas nas aulas.

3.2.1 Tokens simples

Cada um dos lexemas a seguir se confunde com o token a ser retornado, ou seja o token representa apenas um única lexema. A única ação a ser efetuada consiste no retorno do token.

Têm casos especiais ainda mais simples, onde o lexema se limita a um caractere único (exemplos: `*`, ou `;`). Neste caso, o Flex possibilita retornar como constante inteira o próprio caractere (sem necessitar um `#define`, pois será usado um *cast* do *char* de C para um *int*). Por exemplo, se o lexema `*` deve ser reconhecido e associado a um token, ao invês de chamá-lo, por exemplo, de `ASTERISCO` e de programar seu retorno assim:

```
#define ASTERISCO 3333
%%
''*''    { return( ASTERISCO ); }
```

basta escrever:

```
%%
''*''    { return( '*' ); }
```

A tabela a seguir explicita todos os tokens a serem retornados. Para ganhar espaço, alguns casos juntam mais de um token numa linha só, separados por vírgulas. Espera-se bom senso para entender que, por exemplo,

Lexema	Token a ser retornado
`(, )`	’(’, ’)’ (respectivamente)

significa que o lexema `(` deve ser associado ao token ’(’ (exemplo do caso onde o lexema se limita a um único caractere) e que o lexema `)` deve ser associado ao token ’)’, e NÃO que o lexema `(, )` (string composto por “abre-parêntese vírgula fecha-parêntese”) deve ser associado aos dois tokens ’)’ e ’)’ (o que não faria sentido).

Lexemas	Tokens a ser retornados
`int`	INT
`double`	DOUBLE
`float`	FLOAT
`char`	CHAR
`*`,+,−,/	’*’, ’+’, ’-’, ’/’ (respectivamente)
`,`	’,’
`;`	’;’
`'`	QUOTE
`"`	DQUOTE
`(, )`	’(’, ’)’ (respectivamente)
`[, ]`, `{, }`	’`[`’, ’`]`’, ’`{`’, ’`}`’ (respectivamente)
<, >, =	’<’, ’>’ e ’=’ respectivamente
<=	LE
>=	GE
==	EQ
!=	NE
`&&`, `||`, !	AND, OR, NOT (respectivamente)
`if`	IF
`then`	THEN
`else`	ELSE
`while`	WHILE

3.2.1.1 Outros tokens

Em toda essa seção, um dígito é um dos caracteres ’`0`’, ’`1`’, ’`2`’, . . . , ’`9`’.

Descrição	token a ser retornado	Ação
Qualquer combinação de qualquer número de "brancos": espaço branco, tabulação, fim de linha.	não será retornado token	imprimir na saída (tela): `BRANCO`.
Pelo menos uma letra (maíuscula ou minúscula), seguida por qualquer número de letras (maiúsculas ou minúsculas), dígitos ou '_'.	IDF	o lexema deve ser inserido numa tabela de símbolos.
Qualquer conjunto de dígitos	INT_LIT	o lexema, convertido em `int`, deve ser copiado numa variável C global chamada `VAL_INT`, de tipo `int`.
Um número com vírgula flutuante (ver abaixo).	F_LIT	o lexema, convertido em `double`, deve ser copiado numa variável C global chamada `VAL_DOUBLE`, de tipo `double`.

No caso do `F_LIT`, a descrição é a seguinte: qualquer conjunto de dígitos (eventualmente vazio), seguido de um ponto ('`.`'), seguido de pelo menos um dígito. Isso tudo pode ser, opcionalmente, seguido de:

1. `e` ou `E`, obrigatoriamente seguido de
2. um sinal + ou − (obrigatório), obrigatoriamente seguido de
3. pelo menos um dígito.

Exemplos de lexemas reconhecidos: `3.14`, `3.14e+0`, `.0`, `.0E+0`. Exemplos de lexemas não reconhecidos: `0.`, `1e10`, `e+10`, `10.1E`, `.0E0`.

3.3 Tratamento de erros lexicais

Em alguns poucos casos, a entrada pode conter caracteres (ou seqüências de caracteres) não reconhecidos por nenhuma expressão regular. Observa-se que algumas seqüências que parecem, objetivamente, erradas ao programador não podem ser capturadas pelo analizador lexical. Considere por exemplo a entrada: `10E`. Apesar de este lexema não condizer com nada do que se espera, o scanner irá retornar um `INT_LIT` (lexema `10`) e logo depois um `IDF` (lexema `E`). Na verdade, essa entrada é lexicamente correta, e o "erro" que o ser humano detecta é *sintático*: caberá apenas à gramática do analizador sintático não autorizar uma sucessão de dois tokens `INT_LIT` e `IDF`.

Isso dito, existem casos de erros lexicais. Por exemplo, se um caractere '`?`' aparecer na entrada, nenhuma das expressões regulares acima definidas deverá aceitá-lo. Isso acontece também com outros casos. Para pegar tais casos, a solução no Lex é incluir uma última regra (em último lugar para que seja aplicada unicamente em última instância, ou seja se nada fechou antes), do tipo:

```
. { printf(``Erro lexical - caractere nao reconhecido: %c.\n'', yytext[0]);
    exit(-1); }
```

A expressão regular `.` no início da linha significa "qualquer caractere" (único). Ao reconhecer qualquer caractere, então, o analizador lexical gerado pelo Flex irá imprimir na tela uma mensagem relatando o caractere encontrado e não reconhecido, e depois interromper a execução do programa (`exit`) com o

código de retorno negativo, associado a um erro. Observa-se que se usa, no `printf`, a variável interna ao Lex chamada `yytext`. Essa variável, que iremos usar mais adiante, é um string (`char*` em C) que contém sistematicamente o lexema que está sendo comparado com a E. R. Como, nessa régra, se usa a E. R. `.` que autoriza apenas um caractere, `yytext` é aqui um string de tamanho 1, e portanto `yytext[0]` é o único caractere compondo o lexema, que é no caso o caractere que não foi reconhecido. (Para amadores de ponteiros em C, podia se escrever também `*yytext` em lugar de `yytext[0]`.)

O uso de `.` nessa regra deixa claro que se deve aplicá-la apenas como último recurso: é uma regra "pega tudo".

3.4 Observações finais

Vai ser necessário usar a tabela de símbolos da Etapa1. Existe várias opções para incluir o código C (.c e/ou .h) no código do analizador lexical, uma delas sendo usar a mesma técnica usada para incluir `tokens.h`. Deve-se obrigatoriamente incluir um .h que se encontra ou no diretório corrente (no caso: `Pico/src/Etapa2`); ou (melhor) em `Pico/include`, onde o `make install` da etapa 1 terá copiado seus .h anteriores. Da mesma forma, deve-se linkar o código objeto ou com arquivos .o do diretório corrente, ou com os .o de `Pico/objects` (essa segunda opção sendo melhor). Cabe a vocês organizar seu código e seus Makefiles para isso. Os exemplos providos já servem para ilustrar como fazê-lo.

Essa etapa 2 é simples - basicamente, consiste em preencher arquivos existentes com as Expressões Regulares vistas em aula, na sintaxe do Flex. Com os arquivos providos, tudo deve se compilar automaticamente. Posto isso, é muito importante dedicar tempo para *testar* seu analizador lexical e verificar o maior número possível de entradas para verificar que ele retorne os tokens que se espera.

Os testes dessa etapa irão principalmente testar essas expressões regulares.

Chapter 4

Especificação da Etapa 3

4.1 Trabalho pedido

Nessa etapa, você deve programar um analisador sintático (acoplado com seu analisador lexical da Etapa 2) para uma gramática simples, a fim de reconhecer expressões aritméticas.

A gramática a reconhecer é a seguinte:

E $\longrightarrow$ E `'+'` E
E $\longrightarrow$ E `'-'` E
E $\longrightarrow$ E `'*'` E
E $\longrightarrow$ E `'/'` E
E $\longrightarrow$ `'('` E `')'`
E $\longrightarrow$ `'IDF`
E $\longrightarrow$ `INT_LIT`
E $\longrightarrow$ `F_LIT`

O símbolo Start é E. Os terminais são (parte de) os tokens definidos na Etapa 2.

Sentenças como: `IDF '+' '(' IDF '*' '(' F_LIT '-' IDF ')' ')'` são reconhecidas por essa gramática. Posto que o analisador lexical da Etapa 2 esteja funcionado, o programa que você irá entregar nessa etapa deverá poder analisar uma entrada do tipo:
`tmp + (x*( 3.14 - y_0z ))`, reconhecer que essa entrada se decompõe em uma seqüência de tokens tal como a sentença acima, e em torno que essa sentença é sintaticamente correta (ou seja de acordo com a gramática).

Para implementar o analisador sintático, você deverá obrigatoriamente usar o algoritmo LL(1) apresentado na 7a aula (26 de Agosto). Este algoritmo usa, internamente:

- Uma pilha. Você pode usar a pilha implementada na Etapa 1. A pilha deverá armazenar símbolos terminais (tokens) e não-terminais (a definir em função da gramática). Como os terminais retornados por seu analisador lexical são representados através de *int*, sugere-se usar também uma encodificação em *int* de seus símbolos não-terminais.
- Un analisador lexical para ler um por um os tokens da entrada. Isso já vem provido pelo procedimento `yylex()` implementado, graças ao Flex, na Etapa 2 (ver o `main` provido na Etapa 2).
- Uma tabela de decisão, a tabela LL(1). Você deverá montar essa tabela e implementá-la em C para que possa orientar o comportamento do seu parser.

Seu parser chamado `pico` deve aceitar como argumento na linha de comando o nome do arquivo contendo o input (já era o comportamento do seu scanner).

A saída do seu parser, na tela, deve ser a seguinte:

- `OKAY` no caso que a entrada esteja de acordo com a gramática;
- `ERROR` no caso contrário.

Tanto o algoritmo de montagem da tabela LL(1), como seu uso pelo parser, se encontram nas lâminas do encontro 7.

Salienta-se que a gramática acima indicada não é LL(1), por motivos que deveriam aparecer diretamente a quem lembra da aula. A primeira parte dessa etapa consiste em transformar essa gramática numa gramática equivalente, e que seja LL(1).

Para montar a tabela LL(1), será necessário calcular os conjuntos First e Follow de sua gramática transformada. É importante entender que não será preciso programar o cálculo desses conjuntos para qualquer gramática: basta fazê-lo, manualmente, para sua gramática; deduzir a tabela LL(1) que a reconhecerá; e implementar o parser para essa gramática.

4.2 Entrega do trabalho

O diretório `src/Etapa3` de `Pico` deverá incluir, pelo menos:

- um arquivo txt 'gramatica.txt' contendo sua gramática transformada, seguindo o formato seguinte (a título de exemplo, usa-se aqui a gramática acima apresentada):

```
E -> E '+' E
E -> E '*' E
etc...
```

- um arquivo txt 'tabelaLL1.txt' contendo a tabela LL(1), no formato seguinte:

```
Simbolo   '('    ')'    '+'    '-'   '*'   '/'   F_LIT INT_LIT  IDF   $
E       '('E')'                                  F_LIT
etc...
```

  (cada coluna para um símbolo não-terminal em uma linha mostra a derivação a ser aplicada ao não-terminal, quando o próximo token lido é o rótulo da coluna. Por exemplo, aqui, tendo-se `E` no topo da pilha, e lendo-se o token `'('`, deve-se derivar o `E` em `'(' E ')'`.)

- A implementação em C de seu parser LL(1), que deverá incluir (e chamar) o analisador lexical da Etapa 2;
- Um arquivo Makefile que o compila (ver o Makefile de src/Etapa2 para se ajudar).
- Casos de teste.

Chapter 5

Especificação da Etapa 4

Essa etapa consiste na definição da gramática usada no compilador `Pico` e na implementação de um parser que a reconheça, graças à ferramenta Yacc (Bison). Sugere-se (mas não é obrigatório) distribuir partes da gramática entre os três integrantes, cada qual se responsabilizando pela implementação de um pedaço.

Para a programação, pode-se usar o código fonte original liberado no Moodle (pico-etapa4.1.tar.gz). Vem a estrutura de diretórios usual, com Makefiles prontos para serem usados. *A única parte dos Makefiles que pode ser alterada são as regras que dizem respeito ao diretoŕio Pico/Tests/ e às etapas 2 e 3. Não se pode alterar Pico/Makefile, nem Pico/src/Etapa4/Makefile.*

Você *deve*:

- baixar e unzipar pico-etapa4.1.tar.gz;
- copiar em src/Etapa1, src/Etapa2 e src/Etapa3/ os arquivos de suas implementações prévias das 3 primeiras etapas. Para as etapas 2 e 3, podem recuperar também os Makefiles que vocês usaram;
- desenvolver a etapa 4 em src/Etapa4;
- compilá-la a partir de Pico/, com o comando 'make etapa4'.

Os tutores darão suporte no uso dos Makefiles apenas se sua estrutura for respeitada.

5.1 Ajustes no analisador lexical

Foram esquecidos quatro tokens simples a serem acrescentados no scanner da Etapa 2 (arquivo Pico/src/Etapa2/scanner.l): `:`, `END`, `FALSE` e `TRUE`. Basta completar este arquivo com as 4 linhas (na seção correta):

```
":"    {return(':');}
"end"    {return(END);}
"true"   {return(TRUE);}
"false"  {return(FALSE);}
```

Comentário importante. O código fonte entregue (pico 4.1) vem com Makefiles prontos para ajudá-lo a compilar seu parser da etapa 4. Os Makefiles sistematicamente re-aproveitam `Pico/src/Etapa2/scanner.l` para obter o scanner usado pelo Yacc. Ou seja, qualquer modificação que você queira ou precise fazer em seu scanner *deve* ser feita no diretório da Etapa 2, se não ela não será enxergada pela Etapa 4.

5.2 Gramática

Um programa `Pico` é composto por uma seção de declarações, seguida de uma seção de ações (ver o start da gramática no arquivo `pico.y` fornecido).

5.2.1 Declarações

A seção de declarações pode estar vazia, ou é composta por uma ou mais declaração, separada(s) pelo terminal `';'` Uma declaração é composta por uma lista de identificadores, seguida por `':'`, seguido por um indicador de tipo. Uma lista de identificadores consiste em um número qualquer de (pelo menos um) identificador(es) separado(s) por `','`. Um indicador de tipo pode derivar em:

- cada um dos 4 terminais que informa um tipo (ver o capítulo sobre o scanner), ou
- cada um dos 4 terminais que informa um tipo, seguido de `'['`, seguido de uma lista de duplas de inteiros, seguida de `']'`.

  Uma lista de duplas de inteiros é composta por qualquer número (maior ou igual a 1) de ocorrências da seqüência de três símbolos seguintes: `INT_LIT ':' INT_LIT`, cada ocorrência sendo separada da seguinte por `','`.

Exemplo 1 de declarações:

```
x, tmp_1 : double;
```

Exemplo 2 de declarações:

```
x:float;

z1t2     ,     i      :float
```

Exemplo 3 de declarações:

```
x:float;
tab : double[10:20, 30:10, 0:5];
y:float
```

Neste último caso, encontra-se uma lista de três duplas de inteiros: `10:20`, `30:10` e `0:5`.

Comentário. Observa-se que nessa etapa de definição da sintaxe, não se associa semântica ainda a essas construções. No entanto, parece intuitivo que este tipo de declaração servirá para definir um array de 3 dimensões e que cada dupla `n:m` define os valores mínimos e máximos autorizados para o índice em uma direção. Nessa lógica, a segunda dupla `30:10` parece dever levar a um erro, pois se está tentando declarar um array com um limite inferior de índice maior do que o limite superior na segunda dimensão. Repara-se que isso seria um erro semântico, tipicamente detectado na próxima etapa, e não nessa.

5.2.2 Ações

A seção de ações é composta por um ou mais comandos, separados pelo terminal `';'`. Um comando pode ser composto por:

- uma atribuição, do tipo: `lvalue '=' expr`, ou
- um enunciado simples.

Comentário importante. Nota-se que o `;` não faz parte, sintaticamente falando, do comando. Pelo contrário, na linguagem definida aqui, ele é um separador de comandos. Isso significa que seu parser deverá aceitar, eventualmente, um único comando sem ponto-vírgula no fim (ver o exemplo abaixo com o `while() {}`). Isso também significa que não será sintaticamente correta a presença de um ponto-vírgula único (sem comandos a separar).

`lvalue` pode ser ou um identificador simples, ou um identificador seguido por `'['`, seguido por uma lista de `expr`, seguida por um terminal `']'` final. Uma lista de `expr` é composta por pelo menos uma `expr`; se tiver mais de uma, devem ser separadas por `','`.

Uma `expr` abrange:

- duas `expr` separadas por um dos terminais que representa um operador aritmético; ou
- uma `expr` entre parênteses; ou
- um literal único, inteiro ou com vírgula flutuante; ou
- uma `lvalue` tal como definida acima; ou
- uma chamada a um procedimento.

  Uma chamada a procedimento consiste em um identificador, seguido de uma lista de `expr` (se tiver mais de uma `expr` na lista, elas devem ser separadas por `','`) entre parênteses.

Um enunciado simples se limita a uma `expr` ou a uma instrução de controle. Uma instrução de controle pode ser:

- `IF`, seguido de `'('`, seguido de uma expressão booleana, seguida de `')'`, seguido de `THEN`, seguido de um ou vários comando(s) (no sentido definido acima), seguido de:
  - Ou o terminal `END`;
  - Ou o terminal `ELSE`, seguido por um ou vários comandos (no sentido definido acima), seguido(s) pelo terminal `END`.
- `WHILE`, seguido de `'('`, seguido de uma expressão booleana, seguida de `')'`, seguido de `'{'`, seguido por um ou vários comandos (no sentido definido acima), seguido(s) por `'}'`.

Por fim, uma expressão booleana é definida recursivamente da forma seguinte: pode ser:

- Ou o terminal `TRUE`;
- Ou o terminal `FALSE`;
- Ou uma expressão booleana entre parênteses;
- Ou duas expressões booleanas separadas por um dos operadores lógicos reconhecidos pelo scanner (`AND`, `OR`);
- Ou uma expressão booleana precedida pelo token `NOT`;
- Ou duas `expr` separadas por um dos operadores relacionais (tokens: `'<'`, `'>'`, `LE`, `GE` e `EQ`).

Exemplos de ações:

```
x = 1;
f(2-3,x) + tmp1;
if ( g(tmp[1,2]*2) <= 10 ) then
   while ( (x>0) && y!=1  ) { x=0 }
end
```

(Nota-se, nas iterações do `while`, que elas se limitam a apenas um comando `x=0`, portanto sem ponto-vírgula no fim.)

Exemplo de programa `Pico` inteiro:

```
i,n : int;
x:double
x = 2.0; i=0 ; n=10;
while (i <= n) {
   x = x*2 ; i = i+1
}
```

Nota-se a ausência de `;` no fim da última declaração, bem como no fim da última instrução do bloco entre `{` e `}` do `while`. O próprio while, sendo um comando por si só, não vem seguido de `;`.

5.3 Trabalho a ser entregue

A apresentação será feita na terça-feira 30 de Setembro às 10h30. No dia anterior ao encontro, ou seja no dia 29 de Setembro, o código fonte deve ser disponibilizado aos tutores até às 23h00, da forma usual.

Seu parser chamado `pico` deve aceitar como argumento na linha de comando o nome do arquivo contendo o input (já era o comportamento do seu scanner). A saída de `Pico`, na tela, deve ser a seguinte:

- `OKAY` no caso que a entrada esteja de acordo com a gramática;
- `ERROR` no caso contrário.

Sua gramática deve ser diretamente implementada de acordo com a sintaxe do Yacc, dentro de um arquivo chamado `pico.y` no diretório `src/Etapa4`. Encontra-se, no código entregue, um esqueleto de tal arquivo a ser preenchido. Em particular, deve-se especificar os tokens através de `%token` na primeira parte do arquivo `pico.y`, de forma tal que sejam os mesmos tokens usados por seu scanner da Etapa 2.

`Pico` deve usar o yylex provido por seu scanner da Etapa 2. O mais simples para poder usar e compilar seu parser junto com o scanner da Etapa 2 é copiar o arquivo `src/Etapa2/scanner.l` em `src/Etapa4/` e completar o Makefile nesse diretório para obter `lex.yy.c` conforme na Etapa 2 (obs.: como, dessa vez, o `main` está incluído no código do parser, deve-se deletar (ou simplesmente renomear) o `main` que se encontra em `scanner.l`).

Chapter 6

Especificação da Etapa 5

A etapa 5 consiste na geração de código de três endereços ("TAC") para tratar o caso:

- da declaração de variáveis, inclusive de arrays multi-dimensionais;
- das expressões aritméticas, inteiras e com virgula flutuante;
- de algumas verificações semânticas no uso correto de tipos.

6.1 Complementação da Etapa 4

Reaproveitamento de código. É possível[1] que o código entregue nas etapas anteriores tenha, por engano na hora de efetuar um "tar", contido implementações prévias de parte de gramáticas, eventualmente com ações semânticas já escritas, ou ainda com estruturas de dados auxiliares (tabela de símbolos, listas, pilhas...). Se é o caso, você tem liberdade para se inspirar dos mesmos, no entanto: (a) não é nada óbvio que essas implementações sejam melhores do que suas (em particular, elas foram menos testadas...). E (b) uma vez que são herdadas dos dois últimos anos, elas podem não se conformar exatamente ao que está especificado neste semestre.

Sugerimos confiar mais em suas implementações do que tentar reaproveitar esses eventuais pedaços de programas.

Resolução de parte dos conflitos na gramática. O Yacc possibilita indicar a precedência e a associatividade de operadores. Por exemplo, pode-se indicar que o parsing de `2+x+3` deve ser efetuado na ordem `(2+x)+3` (`+` é dito associativo à esquerda), e que `2+x*3` deve ser analizado como `2+(x*3)` (`*` possui precedência maior do que `+`). Essas escolhas possibilitam tirar vários conflitos Shift-reduce que podem ter acontecido em suas gramáticas.

Para isso, basta inserir as linhas abaixo na primeira parte do arquivo pico.y (por exemplo depois das definição dos tokens):

```
%left '+' '-'
%left '*' '/'
%left OR
%left AND
%left NOT
```

[1] Um grupo nos avisou que aconteceu. Não consegui achar estes códigos perdidos nos tar que já liberamos no Moodle...

A ordem das linhas define a precedência crescente dos operadores. Os `left` e `right` definem a associatividade de cada um.

(Em linguagens que possibilitam múltiplas atribuições (`x = y = 1`), é comum incluir uma linha `%right '='` para reduzir na ordem `x = (y=1)`. `Pico` não autoriza isso.)

Nova função na gramática. Pede-se acrescentar a gramática da Etapa 4 com uma nova função, chamada `print`, que aceita um argumento único, que pode ser tanto um número inteiro como um número com virgula flutuante.

6.2 Código TAC a ser gerado

Uma instrução TAC deverá ter o formato seguinte:

```
z := x OP y
```

onde `x`, `y` e `z` deverão ser endereços na memória e `OP` um operador especificado abaixo. Em alguns casos, um dos operandos `x` ou `y` pode estar vazio.

Endereços na memória. Serão representados através de um deslocamento a partir de um endereço base, tipicamente o endereço base de um segmento de pilha. Supor-se-á que esta base se encontra num registrador `SP`, e o endereço na memória será escrito `d(SP)`. Por exemplo, uma variável que for mapeada para o endereço 16 (Bytes) terá como endereço `16(SP)`.

Um tratamento diferente será dado às variáveis temporárias, as quais deverão ser mapeadas através de um deslocamento relativo a um registrador diferente, chamado por convenção `Rx`.

Operadores TAC. Usar-se-á a sintaxe seguinte para os operadores da linguagem TAC:

1. Operações aritméticas com números inteiros se chamarão: `ADD`, `SUB`, `MUL` e `DIV`. Todas terão dois operandos.
2. Operações aritméticas com números em vírgula flutuante se chamarão: `FADD`, `FSUB`, `FMUL` e `FDIV`. Todas terão dois operandos.
3. O símbolo `:=` será usado entre o operando `z` e os dois operandos `x` e `y`, a semântica dessa atribuição sendo que os endereços aparecendo à direita serão derefenciados (usar-se-á o conteúdo dos endereços das variáveis), e que o resultado estará armazenado no endereço encontrado à esquerda. Valores numéricos aparecendo à direita serão considerados como valores imediatos.

   Assim, `z := z+1` significa: consulta a memória no endereço de `z`, soma o valor nele armazenado com 1, e armazena o valor resultante no endereço de `z` (ou seja, curto e grosso: faz o que se espera...).
4. `PRINT`, com um operando único (sintaxe: `PRINT x`)
5. `FPRINT`, com um operando único (sintaxe: `FPRINT x`)

O código TAC inteiro será composto por:

1. um "cabeçote" que informará dois números inteiros, cada um numa linha separada, que definirão repectivamente o espaço total (em Bytes) na memória necessário para armazenar as variáveis manipuladas pelo programa, e (na segunda linha) os temporários.

2. a seqüência de instruções atômicas, cada qual possuindo um número de linha separado da instrução por `:`. O número deverá ser formatado para caber em exatamente 3 caracteres, sendo os números menores do que 100 completados à esquerda com zeros (ou seja, irá de `000` até `999`). O `:` deverá ser seguido por 3 caracteres "espaço branco". Os deslocamentos deverão ser formatados para caber em exatamente três caracteres, também sendo completados à esquerda com zeros se for o caso.

   Para formatar, em C, essas instruções, recomenda-se usar a função `sprintf`. Observa-se que essas cobranças relativas à formatação precisam ser cumpridas na hora de imprimir uma listagem do código TAC, e não obrigatoriamente deve se refletir na representação interna do código.

Exemplo de código TAC esperado:

```
816
16
000:   000(Rx) := 012(SP) MUL 10
001:   004(Rx) := 000(Rx) ADD 008(SP)
002:   008(Rx) := 004(Rx) MUL 4
003:   012(Rx) := 008(Rx)(016(SP))
004:   000(SP) := 012(Rx)
005:   PRINT 000(SP)
```

(Esse trecho de código poderia representar a compilação de `z = A[i,j]; print(z)`, onde `A` seria um array bidimensional de inteiros, tendo 10 elementos na dimensão 1 e 20 na dimensão dois (por isso necessitando $10 \times 20 \times$ 4 Bytes), onde `z` estaria mapeado no endereço 0, `i` no endereço 8, `j` no endereço 12 e `A` teria seu endereço base no endereço 16.)

Nota-se que essa linguagem TAC é de nível mais baixo do que a linguagem usada nas aulas. A fins de depuração, aconselha-se programar primeiramente ações semânticas que geram código TAC com variáveis "simbólicas" (em particular TMP0, TMP1, etc...). Pode-se usar essa primeira versão e a tabela de símbolos, para gerar o código TAC final numa segunda fase. Opcionalmente, pode-se programar um flag `-v` do `Pico` para controlar seu output na forma "TAC alto nível" ou TAC baixo nível. Apenas será exigido o TAC "baixo-nível".

6.3 Código auxiliário para geração de código TAC

Será necessário implementar pelo menos:

1. Uma representação interna de uma instrução TAC. Por exemplo, pode-se usar uma struct de quatro membros para tal.

2. Uma lista encadeada de instruções TAC, junto com métodos para: encadear uma nova instrução na lista; concatenar duas listas; varrer e imprimir o conteúdo da lista sob a forma exigida acima.

3. Um procedimento para gerar "novos temporários".

Nada está imposto sobre essas estruturas de dados, porém será considerado a qualidade da implementação na avaliação. Aconselha-se fortemente testar de forma unitária essa lista.

6.4 Ações semânticas a serem implementadas

6.4.1 Declaração de variáveis

Ao efetuar o *parsing* das declarações (ver o capítulo anterior), deve-se identificar o tipo abstrato de cada variável e inserir na tabela de símbolos, para cada uma delas, uma entrada que represente de forma abstrata:

- O lexema associado ao token `IDF` sendo analizado;
- O tipo da variável;
- O espaço na memória que se deverá usar para armazenar seu valor;
- O seu endereço, ou seja o deslocamento a partir do endereço base aonde se encontrará mapeada a variável;
- Informações extras no caso de um *array* (ler mais adiante).

No `Pico`-2008, tem-se apenas um escopo único e global para todas as variáveis, e por isso se usará apenas uma tabela de símbolos global. *Todas as variáveis devem ter sido declaradas antes de serem usadas no programa `Pico`. Uma variável referenciada sem ter sido declarada previamente deverá levar seu compilador a disparar um erro* `UNDEFINED_SYMBOL`

Um erro `UNDEFINED SYMBOL` deverá ser provocado, em C/Yacc, da forma seguinte:

```
#define UNDEFINED_SYMBOL_ERROR -21
/* muitas linhas de codigo, pois o define acima esta lah no topo do pico.y */
/* ... */
XXX: YYY {
    printf(``UNDEFINED SYMBOL. A variavel %s nao foi declarada.\n'', lexema);
    return( UNDEFINED_SYMBOL_ERROR );
    }
```

(Onde `XXX`, `YYY` e o string `lexema` estão usados de forma genérica, obviamente. Cabe-lhes identificar as ações onde se deve usar tal disparo de erro.)

O lexema se encontra, no scanner, na variável `yytext`, atualizada pelo scanner com o Flex. Lembra-se que do lado do Yacc, a variável `yylval` é usada para o atributo.

Os tipos básicos terão o tamanho seguinte:

- tipo char: 1 Byte;
- tipo int: 4 Bytes;
- tipo float: 4 Bytes;
- tipo double: 8 Bytes.

O deslocamento e tamanho da variável deverão ser calculados à medida que o *parsing* anda, conforme explicado em aula. Atributos semânticos deverão ser associados aos símbolos não-terminais e terminais apropriados, com o Yacc, da forma descrita abaixo (e já apresentada em aula).

Arrays. No caso dos arrays, o trabalho de representação do tipo pode ser mais complexo. No `Pico`, iremos nos limitar ao caso de arrays de tipos simples. A representação de arrays (de arrays)* de arrays[2] necessita o uso de um grafo abstrato que não vai ser implementado neste semestre. Para arrays de tipos simples, deverá ser reservado o espaço na memória igual ao número total de elementos no array, multiplicado pelo tamanho do tipo simples (em Bytes). Na tabela de símbolos, uma entrada associada a um array deverá, obviamente, informar este tamanho, e também conter os valores pre-calculados dos elementos necessários ao cálculo semântico do acesso a um elemento indexado no array (ver a aula sobre arrays multidimensionais).

Assim, a `struct entry_t` preparada na Etapa 1, que continha um campo `void* extra`, deverá ser usada e complementada para que este campo se torne um ponteiro para uma nova struct, com todas as informações necessárias (por exemplo: a constante "c"; o número de dimensões; os limites inferiores e superiores em cada uma das dimensões; etc...). Caberá à implementação consultar esses campos, quando há necessidade numa ação semântica.

Observa-se que vocês têm uma certa margem de manobra nessa representação interna de seus tipos, em especial no caso dos arrays.

Ao terminar a análise sintática da declaração de variáveis, o tamanho total usado na memória para todas as variáveis deverá ser armazenado, para depois ser impresso junto com o código TAC no cabeçalho.

6.4.2 Expressões aritméticas

Todas as ações semânticas necessárias à geração de código para expressões aritméticas devem ser implementadas. Entende-se, nessa etapa, por "expressão aritmética", todas as derivações possíveis, tais como definidas na Etapa 4, a partir de uma atribuição do tipo `lvalue '=' expr` (ver Sec. 5.2.2, p. 20), SENDO EXCLUÍDAS as derivações de `expr` em chamada a um procedimento. Informalmente, isso significa que se espera qualquer combinação de operadores aritméticos atuando em identificadores ou arrays, os quais podem aparecer tanto à esquerda como à direita de um sinal de atribuição.

Não é necessário alterar sua gramática — basta implementar as ações semânticas relativas às produções que dizem respeito à sub-gramática tratada nessa Etapa. Outras virão nas Etapas 6 e 7.

O fato de incluir os identificadores nas expressões, assim como os arrays, significa que se deverá efetuar as ações semânticas relevantes de consulta à tabela de símbolos, ao encontrar um `IDF`, para verificar se foi declarado, e se sim recuperar as informações necessárias à geração de código (se é relevante).

No caso de uma expressão derivar em um terminal `F_INT` *ou* `INT_LIT`*, deve-se usar o próprio lexema na instrução TAC gerada. Por exemplo, ao compilar* `z = y + 3.14e-1`*, o código TAC gerado irá conter uma linha com* `3.14e-2` *explicitamente escrito (numa instrução ADD).*

6.4.3 Instrução print

Deve-se gerar código TAC, através de ações semânticas, para o único caso de chamada a procedimento que é a chamada a `print`. A(s) ação(ões) semântica(s) deverá(ão) efetuar checagem do tipo do argumento para gerar o código apropriado em função do número.

6.4.4 Checagem de tipos

Conforme especificado acima, a linguagem TAC diferencia operações em números inteiros das operações em vírgula flutuante. Isso significa que você deverá incluir em sua análise semântica a determinação do tipo dos operandos de um operador, para poder:

[2] Para quem não entendeu o asterisco, isso era uma expressão regular...

1. Verificar a compatilidade entre os tipos dos operandos e o operador;
2. Gerar o código apropriado ao tipo dos operandos.

Sobre o primeiro ponto, deve ser previsto que apenas "números" podem ser alvo de um operador aritmético e de um `print`. Por exemplo, uma tentativa de somar um `char` com um `int` deve levar a um erro (TYPE_MISMATCH_ERROR). Faz parte do trabalho formalizar o que será um "número" (em nível semântico)[3].

Um TYPE_MISMATCH_ERROR deverá ser provocado, em C/Yacc, da forma seguinte:

```
#define TYPE_MISMATCH_ERROR -20
/* muitas linhas de codigo, pois o define acima esta lah no topo do pico.y */
/* ... */
expr: uma coisa MUITO complexa {
    printf(``ERRO DE COMPATIBILIDADE DE TIPO. Quem te ensinou a programar?\n'');
    return( TYPE_MISMATCH_ERROR );
    }
```

No que diz respeito à geração de código, deve ser previsto:

1. que uma expressão será considerada como inteira se todos seus operandos possuem o tipo `int`.
2. que tão logo um operador aritmético será aplicado a pelo menos um "número" em vírgula flutuante, deverá ser criada uma variável temporária de tipo `float`, à qual será atribuído o valor do eventual operando inteiro. O operador será, então, aplicado ao temporário e ao operando não-inteiro.
3. Que o acesso a um elemento de um array multidimensional só pode ser efetuado pelo uso de uma lista de expressões cujo tipo seja inteiro. Qualquer outro caso deve levar ao erro TYPE_MISMATCH_ERROR. Exemplos disso:

   ```
   k: double
   x = a[1,3.14];
   b[k] = 0
   ```

   Neste exemplo, têm dois erros semânticos.
4. Que a instrução TAC a ser gerada ao parsar `print` deve ser diferenciada em função do argumento ser inteiro ou com vírgula flutuante.

6.5 Atributos no Yacc

(Consultar também a apostilha Yacc disponbilizada no Moodle, em baixo à direita da página.)

Os atributos semânticos no Yacc têm tipo inteiro por *default*. Para qualquer regra genérica na gramática:

```
L: A B C D E
 ;
```

O atributo associado ao símbolo de esquerda (`L`) se chama `$$`, o atributo do primeiro símbolo (não-terminal ou terminal) à esquerda (`A`) se chama $1, o seguinte é $2 (no caso para `B`), etc. Por exemplo,

[3] Essa frase pode parecer ameaçadora, mas apenas significa que vocês devem implementar ações semânticas (simples) nas regras que lhes parecem manipular números.

```
L: A B C D E  { $$  = $3+$5; }
 ;
```

levará a sintetizar o atributo de `L` como sendo igual à soma dos atributos de `C` e `E`. No caso, pode-se efetuar a soma, pois todos os atributos têm o tipo default inteiro.

Outro *default* importante é que para uma regra do tipo:

```
L: R;
```

que se encontra frequentemente, há implicitamente a ação semântica `{$$ = $1}`.

Toda a dificuldade vem do fato de se querer usar atributos não-inteiros, e eventualmente mais de um atributo por símbolo. Isso se faz da forma seguinte. Vamos supor que se queira usar dois atributos para um símbolo não-terminal N, um chamado size, de tipo int; e um chamado type, de tipo void*. E vamos supor que um outro símbolo T, terminal dessa vez (ou seja: um token), vá precisar de um atributo de tipo char*, chamado name.

Declara-se esses atributos assim:

```
%union {
   struct { int len ; void* type ; } abstract_att;
   char* name;
}

%token<name> T
%type<abstract_att> N
```

A `union` possibilita a descrição dos tipos de atributos que serão usados, e sua associação a um nome (`abstract_att` e `name`). Esses nomes (de tipo de atributos) são usados, opcionalmente, na instrução `%token` já encontrada, e na instrução `%type`, para especificar um tipo não inteiro do atributo $x de um símbolo token ou não-terminal.

Mais adiante, o usuário pode então usar esses atributos, por exemplo da forma seguinte na regra totalmente imaginária:

```
N: T N ';' T  { $$.type  = $1; $$.len = strlen($4); }
 ;
```

Que faz essa regra? Em primeiro lugar, observa-se que como $$ é o atributo de `N`, ele é de tipo `struct { int len ; void* type ; }`. Por isso, pode-se acessar seus campos `type` (void*) e `len` (int). Da mesma forma, uma vez que `T` tem atributo de tipo `char*`, tanto $1 (atributo do primeiro símbolo à direita — que é `T`) como $4 (observa-se que $3 é o atributo de `';'`) possuem tipo `char*`. Assim sendo, pode-se calcular `strlen($4)`, ou usar cópia entre ponteiros para escrever `$.type  = $1`.

Chapter 7

Especificação da Etapa 6

A etapa 6 consiste na geração de código de três endereços ("TAC") para tratar o caso:

- do comando if... then... else;
- do comando while.

7.1 Precisões sobre as Etapas 5 e 6

Nessa seção, uma série de precisões são dadas a respeito do `Pico`.

Números negativos. A especificação dos tokens que representam números (Etapa 2) não inclui números negativos. Isso foi proposital para evitar ambigüidades entre o menos unário (-3.14) e o menos que denota a subtração ($1.0 - 3.14$). Existe uma forma de resolver tal ambigüidade com o Yacc (basta aumentar a prioridade do menos unário em comparação à prioridade do menos binário). Como não se admite, em nível lexical, número negativo, tal problema não é tratado no `Pico`.

Invocação do `Pico`. A partir da Etapa 6, impõe-se o uso seguinte de `Pico`: `pico -o <filename> <input>.pico`, onde:

- `<input>.pico` pode ser qualquer nome de arquivo que contenha código fonte `Pico`. A extensão `.pico` é obrigatório e `Pico` não deve aceitar compilar um arquivo de entrada que tenha outra extensão.
- `<filename>` é o nome do arquivo de saída, que deve conter código TAC válido, e apenas código TAC. Este nome pode ser qualquer nome de arquivo válido, com qualquer extensão. Deve conter apenas TAC válido, ou seja não deve constar nele linhas tipo "OKAY", ou "ERROR". Este arquivo de saída será compilado de TAC para linguagem de montador, por isso deve ser rigorosamente de acordo com a síntaxe TAC definida na Etapa 5.

Exemplos de uso:

```
pico -o test1.tac test1.pico
pico -o output input.pico
pico -o resultado.txt prog_1_.pico
```

7.2 Ações semânticas

7.2.1 Expressões booleanas

A avaliação das expressões booleanas, necessárias para o controle das instruções `if` e `while`, deverá ser feita com código do tipo "curto-circuito" e levar ao valor correto dos atributos `falso` e `verda` dessas expressões. Para isso, você irá precisar de um novo procedimento chamado `gera_rotulo()` que irá gerar a cada chamada um novo rótulo simbólico; um tal rótulo será uma seqüência de caracteres quaisquer, seguidos por dois pontos (`:`). Por exemplo, `gera_rotulo` pode retornar sucessivamente: `label0:`, `label1:`, `label2:`, etc...

Para as ações semânticas necessárias, ver a aula do dia 09 de Outubro.

7.2.2 Comandos de controle de fluxo

Você deverá usar as ações semânticas que foram apresentadas na aula do dia 09 de outubro, com os devidos atributos herdados para o cálculo dos atributos dos símbolos relativos às expressões booleanas.

Para o uso de atributos herdados com o Yacc, lembra-se que a sessão de laboratório da terça-feira 07 de Outubro incluiu material sobre o uso de uma pilha de atributos.

7.3 Código TAC

O código TAC produzido na Etapa 6 deve ser idêntico ao da Etapa 5, extendido com os seguintes comandos:

LABEL.

```
<label>:
```

onde `<label>` é um identificador de rótulo válido. Um indicador de rótulo válido começa com uma letra (obrigatoriamente maiúcula) seguida ou não de qualquer combinação de letras maiúsculas, dígitos e *underscores*. Esse comando indica que a parte do código subseqüente está endereçada através de um comando `GOTO`. Exemplo de identificador de rótulo:

```
MEU_LABEL_2:
```

Atente que **não deve haver número da linha ou espacos na instrução**.

GOTO. A instrução `GOTO`:

```
xxx:   GOTO <label>
```

onde `xxx:` é um número de linha qualquer (como a Etapa 5, deve ser suscedido de exatamente 3 espaços) e `<label>` é um rótulo definido como acima. Esse comando ajusta o fluxo de instruções para que passe a ser executado logo após (na linha seguinte a) um rótulo. *e.g.*,

```
000:   GOTO COMECA_AQUI
001:   004(Rx) := 000(Rx) ADD 008(SP)
COMECA_AQUI:
002:   008(Rx) := 006(Rx) ADD 4
```

nesse exemplo, a linha `001` nunca é executada.

Opcionalmente, o comando `GOTO` pode referenciar explicitamente uma linha no código TAC, dispensando a definição do rótulo. Nesse caso, o número da linha deve ser precedido de um *underscore*. O seguinte código faz o mesmo que o anterior:

```
000:   GOTO _002
001:   004(Rx) := 000(Rx) ADD 008(SP)
002:   008(Rx) := 006(Rx) ADD 4
```

(Observa-se que, através de um `GOTO _001`, se pode executar a linha `001` afinal.)

IF. O comando `IF` é implementado como abaixo. Repare que o número de espaços entre os componentes deve ser respeitado (como sempre)!

```
xxx:   IF <op1> <comp> <op2> GOTO <label>
```

onde:

xxx é um número de linha qualquer.

op1 é um operando, que pode ser ou um número em valor absoluto ou um número armazenado na memória (como a Etapa 5, com referência aos registradores Rx e SP).

comp é um comparador booleano. São admitidos os seguintes comparadores:

- <
- >
- <=
- >=
- ==
- !=

op2 é como op1.

label é como o rótulo definido para a instrução GOTO (aceitando, inclusive, o número da linha, precedido por um *underscore*, conforme explicado anteriormente).

e.g.,

```
000:   IF 4 <= 008(SP) GOTO MEU_LABEL
001:   004(Rx) := 000(Rx) ADD 008(SP)
MEU_LABEL:
002:   008(Rx) := 006(Rx) ADD 4
```

executa a linha `001` apenas quando a comparação da linha `000` resultar em verdadeiro.

Chapter 8

Especificação da Etapa 7

Nessa etapa, você deve desenvolver um programa que traduz o código TAC em código *assembly* x86. O mesmo será montado e executado.

8.1 TAC vs. *Assembly* x86

A conversão de código TAC em *assembly* se faz como segue.

8.1.1 Pré-Requisitos

Para transformar o programa gerado em código de máquina, é necessário um ambiente Linux com os seguintes programas instalados e funcionais:

1. GNU Assembler ("`as`")
2. GNU Linker ("`ld`")
3. GNU libc ("`glibc`")

em geral, a maioria das distros Linux (*e.g.*, Ubuntu) vêm com esses programas instalados.

8.1.2 O *Script* a ser programado

O trabalho desta Etapa consiste em construir um script chamado `pias` – PIco ASsembler – que, dado um código TAC gere o correspondente *assembly* x86. O seu *script* deverá ser invocado da seguinte maneira, em *bash script*:

```
pias <nome_arquivo.tac>
```

Essa chamada deverá fazer o seguinte:

1. Chamar um tradutor em cima do arquivo passado como parâmetro, que transforma o código TAC em um código *assembly*. O arquivo resultante deve se chamar "output.s"
2. Chamar o GNU Assembler sobre o arquivo "output.s", resultando no arquivo "output.o", através do comando seguinte:

```
as output.s -o output.o
```

3. Chamar o GNU Linker sobre o arquivo "output.o", gerando como saída o arquivo "output", que é um executável válido. Deve-se ligar o código à `glibc`, pra que se possa usar a função `printf` dentro do código assembly. Isso se faz através do comando:

   ```
   ld -dynamic-linker /lib/ld-linux.so.2 -o output -lc output.o
   ```

   (O diretório `/lib/ld-linux.so.2` é a localização da `glibc` para o Ubuntu, podendo variar para outras distribuições. Consulte a documentação própria.)

Basicamente, isso significa que seu script `pias` será composto por três linhas, cada qual efetuando uma das etapas acima descritas. A real complexidade da tarefa será implementar a etapa (1) (tradutor), conforme descrito a seguir.

8.1.3 Implementação do Tradutor

Um tradutor é um analisador de expressões regulares que irá ler uma linha de instrução TAC (no formato descrito nas etapas anteriores) e transformá-la em algumas linhas *assembly* x86. Não é necessário um Compilador (análise sintática e semântica), pois a especificação do TAC foi feita em cima de uma Linguagem Regular, propositadamente.

O tradutor pode ser implementado da maneira que o aluno preferir, escolhendo, inclusive, em que linguagem deseja implementar. A única restrição é que ele deve funcionar em alguma das máquinas do laboratório usado para as aulas práticas (verifiquem!). Pode-se usar o (F)LEX, por exemplo. Entretanto, salienta-se que será provavelmente mais simples programar o tradutor com uma linguagem como Java, Python, Perl ou Ruby do que com o (F)LEX.

Para facilitar a tarefa, um arquivo ("model.template") de modelo será fornecido. Esse arquivo já está pronto para ser processado pelo `as`, faltando apenas preencher as seguintes lacunas:

1. `<stack_size>` : o tamanho da pilha. É o número da primeira linha do código TAC.
2. `<temp_size>` : tamanho reservado às variáveis temporárias. É o número na segunda linha do código TAC.
3. `<code>` : é o código *assembly* equivalent ao código TAC.

Recomenda-se copiar o template linha-a-linha para o arquivo "output.s", substituindo nele as marcações `<stack_size>`, `<temp_size>` e `<code>` com os dados requeridos, obtidos pela análise do TAC.

Para a geração de código assembly, será fornecida documentação de todos os programas usados, bem como a linguagem *assembly* x86 usado no montador `as`. O aluno deve relembrar a arquitetura usada pelos processadores Intel (registradores, endereçamento, etc.) e pode usar esses documentos para tal. Faz parte da atividade pesquisar sobre a arquitetura-alvo, como faria um projetista do compilador. Uma série de exemplos de conversões será fornecido para ajudar essa fase, na forma de uma par de arquivos, um com o código TAC e outro com o respectivo *assembly* gerado.

Geralmente, uma instrução TAC é convertida para $n > 1$ instruções assembly, pois há necessidade de salvar registradores, usá-los e restaurar seu conteúdo. *e.g.*,

```
003:   012(Rx) := 008(Rx) ADD 016(SP)
```

é convertida em

```
_003: MOVL  8(%EDX) , %EAX
      ADDL  16(%ECX) , %EAX
      MOVL  %EAX ,12(%EDX)
```

A instrução TAC `print` será implementada através de uma chamada ao procedimento `printf` da linguagem C da seguinte maneira:

```
005:   PRINT 004(SP)
```

é convertida em

```
_005: PUSHL  4(%ECX)
      PUSHL  $intf
      CALL printf
```

seu trabalho será apenas traduzir `004(SP)` em `4(%ECX)`, nesse caso. Para saber que registradores da arquitetura correspondem a SP e Rx, veja o arquivo "template.model".

A conversão do número de linha TAC para o código *assembly* é como mostrado anteriormente. Não será cobrada indentação para o código *assembly*. O cerne da avaliação será o executável gerado e sua correspondência com a linguagem processada pelo `pico`. Isso não significa, entretanto, que o código gerado não será avaliado.

8.1.4 Limitações & Facilidades

O emprego de números em ponto flutuante **não** será cobrado nessa etapa, tratando apenas operações com números inteiros. A avaliação de expressões booleanas também não precisará ser curto-circuito.

8.2 Data de entrega

A etapa 7 deverá ser entregue até o dia 01 de Dezembro, 23h00. O script `pias` e os demais executáveis deverão se encontrar no diretório `Pico/src/Etapa7`.

Chapter 9

Avaliação

A avaliação se fará em duas etapas.

Primeiramente, seu programa irá ser testado de forma automatizada: programas C pre-escritos irão ser compilados junto com suas implementações de tabela de símbolos e de pilha, e executar operações usuais nelas para verificar sua conformidade à especificação (exemplos de tais testes serão providos nessa semana). A conformidade irá levar a 50% da nota final obtida nessa etapa 1. Salienta-se que, nessa fase, programas de verificação de cópia e programas de busca por código on-line serão usados para detectar eventuais fraudes.

A outra metade da nota será atribuída em função da apresentação informal que vocês farão em laboratório, no dia 25 de Setembro, ao seu tutor: resultados dos testes, discussão da implementação, validação através de outros testes, documentação. Todos os três alunos podem ser indagados sobre qualquer parte da implementação, até mesmo quem não programou uma funcionalidade: espera-se que todos os integrantes tenham entendido como tudo foi programado. Opcionalmente, o tutor poderá aplicar um questionário rápido, e escrito, para testar o conhecimento do programa entregue.

Qualquer tentativa, mesmo parcial, de reaproveitamento de código de colegas ou da Web, irá zerar a nota.

Chapter 10

Anexo técnico: Doxygen e Make

10.1 Introdução ao DOXYGEN

Todo o conteúdo deste tutorial foi retirado de http://www.stack.nl/~dimitri/doxygen/manual.html. Recomenda-se consultar a fonte para mais detalhes. O que segue são as notações recomendadas para o projeto da disciplina de Compiladores (INF01147/INF01033) 2008/2.

Gerador de documentação para código-fonte, o DOXYGEN é um sistema de documentação para C++, C, Java, Objective-C, Python, *etc.*

Dentre outras capacidades, o DOXYGEN pode gerar um sistema de documentação *on-line* (em HTML) e/ou um manual de referência *off-line* (em LaTeX) de um conjunto de códigos-fonte propriamente documentados. Também há suporte para gerar a saída em RTF (*MS-Word*™), *PostScript*, PDF com *hyperlinks*, HTML compactado e páginas *man* do UNIX. A documentação é extraída diretamente dos códigos-fonte, tornando mais fácil a manutenção da documentação. Disponível para Windows™, Linux e MacOS-X™.

10.1.1 Comandos Básicos em C

Existem muitos métodos diferentes para escrever um comentário no estilo DOXYGEN . Listados, a seguir, os mais comuns.

10.1.1.1 Blocos de Comentários Simples

Comentários simples na sintaxe da linguagem podem ser colocados normalmente. Por padrão, o DOXYGEN **não** gerará estes comentários na saída, ignorando sua presença. São necessários identificadores especiais para os comandos DOXYGEN .

```
/* comentario simples */
```

10.1.1.2 Blocos de Comentários Breves

Um dos modos em que o DOXYGEN pode gerar comentários, estes são produzidos na saída; devem possuir apenas uma linha, são usadas para considerações pontuais importantes (como a descrição do que faz um laço `for` complexo) e sua sintaxe é a seguinte:

```
/** comentario breve */
```

Também pode ser obtido através da *tag* `\brief`:

```
/* \brief comentario breve */
```

10.1.1.3 Blocos de Comentários Detalhados

Um dos modos em que o DOXYGEN pode gerar comentários, estes são produzidos na saída; usados para descrições detalhadas de blocos dos arquivos, em várias linhas (como o *header* do código fonte ou cabeçalho de função). Sua sintaxe é a seguinte:

```
/**
 * comentario detalhado
 *
 */
```

10.1.1.4 Tags Especiais

Algumas *tags* (marcações especiais do código que aparecem an documentação final) são de especial uso neste projeto:

author : nome autor do autor do código em questão (vários autores de vários `author` serão agrupados no mesmo parágrafo).

version : versão atual do código (*e.g.*, número da etapa do projeto).

date : data da primeira e última modificação no código.

bug : descreve possíveis *bugs* que a aquele trecho de código apresenta.

warning : aviso no uso daquele trecho de código.

param : identifica o parâmetro de uma função. Deve ser usado com as palavras-chave *in* e *out* para indicar o tipo de atributo (entrada ou saída) e uma breve descrição.

return : valor de retorno de uma função.

file : identifica o arquivo (nome dele no sistema de arquivos) ao qual o comentário se refere. Em geral é o nome do arquivo de código-fonte e aparece apenas no *header*.

Por exemplo,

```
/**
 * \file minhas_funcoes.c
 * \brief Arquivo com as minhas funcoes.
 */

/** MinhaFuncao
 *  \brief Copia sequencias de bytes.
 *  \author Bill Gates
 *  \author Linus Torvalds
 *  \version 4.0
 *  \date    1996-1998
 *  \bug Trava muito e requer muita memoria.
 *  \bug A medida que eh usada introduz mais bugs.
```

```
 *  \warning Nao confie nesta funcao.
 *  \warning Nem neste comentario.
 *
 * Nao faz nada e soh consome memoria.
 * \param[out] dest Area de destino.
 * \param[in]  src  Area de origem.
 * \param[in]  n    Numero de bytes.
 * \return Nada.
 */
void minha_funcao(void *dest, const void *src, size_t n);
```

10.1.1.5 Uso

O DOXYGEN consegue determinar a que bloco de código um comentário se refere. Isso significa que blocos que antecedem uma função referem-se à função e blocos que antecedem todo o arquivo são o *header* do arquivo na documentação. Neste projeto, três blocos são o suficiente para que o arquivo seja considerado documentado: o *header*, o descritor da função e comentários de trechos importantes de código. Em anexo, segue um exemplo.

10.2 Introdução ao Make

10.2.1 Introdução

Make é um programa GNU linux que automatiza o procedimento de criação e manutenção de grupos de programas, verificando dependências e possibilitando a compilação de códigos. O objetivo de make é determinar automaticamente as partes de um programa relativamente grande que precisam ser recompiladas, provendo os comandos necessários para tal finalidade. Com make podem ser descritas muitas tarefas onde arquivos precisam ser atualizados automaticamente a partir da modificação de outros.

Em um programa, o arquivo executável é obtido a partir de arquivos objeto, os quais são oriundos da compilação de arquivos fontes. Com apenas uma única chamada de make, todas as mudanças realizadas em arquivos fonte podem ser recompiladas, uma vez que make se baseia nos tempos das últimas modificações do arquivo. Caso não haja nenhuma alteração, não será necessária a compilação, não havendo a necessidade de recompilação de todo código

10.2.2 Composição de um arquivo makefile

Para usar make, é necessário criar um arquivo chamado *makefile* que descreve as relações entre os arquivos do programa e os estados dos comandos para a atualização de cada arquivo. Em um arquivo makefile podem ser utilizados *labels* (rótulos) e *targets* (alvos), de maneira que na chamada de make possa haver o reaproveitamento de nomes, bem como a invocação de diferentes alvos. Também é possível verificar dependências. Por exemplo, se o arquivo makefile tiver a seguinte especificação:

```
FILENAME=fonte
FLAGS= -g -Wall

compilar:
        ${CC} -o ${FILENAME} ${FILENAME}.c ${FLAGS}

executar: ${FILENAME}
        ./${FILENAME}
```

```
apagar:
        rm ${FILENAME}.o
        rm ${FILENAME}
clear
```

Tem-se um *label*, $FILENAME$, o qual está referenciando a palavra "fonte", sendo usado em diferentes partes do script como uma "variável" e a *label* FLAGS, o qual é invocada apenas em uma das *targets*

Ao mesmo tempo, têm-se três *targets* que podem ser chamados através de:

```
nome@maquina:/home/usuario: make compilar

nome@maquina:/home/usuario: make executar

nome@maquina:/home/usuario: make apagar
```

No primeiro caso haverá a compilação do código utilizando o compilador gcc. Este é invocado através de uma variável externa. Já as *flags* de documentação estão referenciadas em *FLAGS*.

No segundo caso a *target* verifica a existência do código executável (repare que isto é feito logo após os dois pontos) e se isto for verdadeiro ele executa o código.

Já no terceiro caso haverá a execução de três comandos: a exclusão do arquivo objeto, a exclusão do arquivo executável e a "limpeza" do terminal.

Para o acesso ao conteúdo de um *label* basta utilizar $ antes do *label*, sendo esta colocada entre chaves. Outra observação importante é de que os comandos precisam ser colocados a partir da linha seguinte de *target*, iniciando sempre com um *tab*.

É através da invocação dos targets que se pode executar uma série de comandos práticos, simplificando o processo de compilação.

10.2.3 Maiores Referências

Sugere-se a consulta de:

$man\ make$

$http://www.gnu.org/software/make$

$http://www.gnu.org/software/make/manual/make.html$